IRIX User's Reference Manual

Volume I

Version 5.1

Document Number 007-0606-050
(Includes *IRIX User's Reference Manual Update,*
Document Number 007-0606-051)

5/90

IRIX User's Reference Manual, Volume I
Version 5.1
Document Number 007-0606-050
(Includes *IRIX User's Reference Manual Update,* Document Number 007-0606-051)

Silicon Graphics, Inc.
Mountain View, California

TABLE OF CONTENTS

1. Commands

NAME

intro – introduction to commands, application programs, and programming commands.

DESCRIPTION

This section describes, in alphabetical order, publicly-accessible commands. Certain distinctions of purpose are made in the headings:

(1)	Commands of general utility.
(1C)	Commands for communication with other systems.
(1G)	Graphics utilities.

Manual Page Command Syntax

Unless otherwise noted, commands described in the **SYNOPSIS** section of a manual page accept options and other arguments according to the following syntax and should be interpreted as explained below.

name [*–option* ...] [*cmdarg* ...]
where:

[ ]	Surround an *option* or *cmdarg* that is not required.
...	Indicates multiple occurrences of the *option* or *cmdarg*.
name	The name of an executable file.
option	This is either *noargletter* ... or *argletter optarg* [,...] It is always preceded by a "–".

noargletter
A single letter representing an option without an option-argument. Note that more than one *noargletter* option can be grouped after one "–" (Rule 5, below).

argletter
A single letter representing an option requiring an option-argument.

optarg
An option-argument (character string) satisfying a preceding *argletter*. Note that groups of *optargs* following an *argletter* must be separated by commas, or separated by white space and quoted (Rule 8, below).

cmdarg
Path name (or other command argument) *not* beginning with ‘‘–’’, or ‘‘–’’ by itself indicating the standard input.

Command Syntax Standard: Rules

These command syntax rules are not followed by all current commands, but all new commands will obey them. *getopts*(1) should be used by all shell procedures to parse positional parameters and to check for legal options. It supports Rules 3-10 below. The enforcement of the other rules must be done by the command itself.

1. Command names (*name* above) must be between two and nine characters long.
2. Command names must include only lower-case letters and digits.
3. Option names (*option* above) must be one character long.
4. All options must be preceded by ‘‘–’’.
5. Options with no arguments may be grouped after a single ‘‘–’’.
6. The first option-argument (*optarg* above) following an option must be preceded by white space.
7. Option-arguments cannot be optional.
8. Groups of option-arguments following an option must either be separated by commas or separated by white space and quoted (e.g., `–o xxx,z,yy` or `–o "xxx z yy"`).
9. All options must precede operands (*cmdarg* above) on the command line.
10. ‘‘– –’’ may be used to indicate the end of the options.
11. The order of the options relative to one another should not matter.
12. The relative order of the operands (*cmdarg* above) may affect their significance in ways determined by the command with which they appear.
13. ‘‘–’’ preceded and followed by white space should only be used to mean standard input.

Throughout the manual pages there are references to *TMPDIR*, *BINDIR*, *INCDIR*, *LIBDIR*, and *LLIBDIR*. These represent directory names whose value is specified on each manual page as necessary. For example, *TMPDIR* might refer to /tmp or /usr/tmp. These are not environment variables and cannot be set. (There is also an environment variable called **TMPDIR** which can be set. See *tmpnam*(3S).)

SEE ALSO

getopts(1), exit(2), wait(2), getopt(3C).

DIAGNOSTICS

Upon termination, each command returns two bytes of status, one supplied by the system and giving the cause for termination, and (in the case of ''normal'' termination) one supplied by the program (see *wait*(2) and *exit*(2)). The former byte is 0 for normal termination; the latter is customarily 0 for successful execution and non-zero to indicate troubles such as erroneous parameters, or bad or inaccessible data. It is called variously ''exit code'', ''exit status'', or ''return code'', and is described only where special conventions are involved.

WARNINGS

Some commands produce unexpected results when processing files containing null characters. These commands often treat text input lines as strings and therefore become confused upon encountering a null character (the string terminator) within a line.

A – B

NAME

acctcom – search and print process accounting file(s)

SYNOPSIS

acctcom [[options] [file]] . . .

DESCRIPTION

acctcom reads *file*, the standard input, or */usr/adm/pacct*, in the form described by *acct*(4) and writes selected records to the standard output. Each record represents the execution of one process. The output shows the **COMMAND NAME, USER, TTYNAME, START TIME, END TIME, REAL (SEC), CPU (SEC), MEAN SIZE(K)**, and optionally, *F* (the *fork/exec* flag: **1** for *fork* without *exec*), **STAT** (the system exit status), **HOG FACTOR, KCORE MIN, CPU FACTOR, CHARS TRNSFD**, and **BLOCKS R/W** (total blocks read and written).

A # is prepended to the command name if the command was executed with superuser privileges. If a process is not associated with a known terminal, a ? is printed in the **TTYNAME** field.

If no *files* are specified, and if the standard input is associated with a terminal or */dev/null* (as is the case when using & in the shell), */usr/adm/pacct* is read; otherwise, the standard input is read.

If any *file* arguments are given, they are read in their respective order. Each file is normally read forward, i.e., in chronological order by process completion time. The file */usr/adm/pacct* is usually the current file to be examined; a busy system may need several such files of which all but the current file are found in */usr/adm/pacct?*

OPTIONS

–a	Show some average statistics about the processes selected. The statistics will be printed after the output records.
–b	Read backwards, showing latest commands first. This *option* has no effect when the standard input is read.
–f	Print the *fork/exec* flag and system exit status columns in the output. The numeric output for this option will be in octal.
–h	Instead of mean memory size, show the fraction of total available CPU time consumed by the process during its execution. This "hog factor" is computed as: (total CPU time)/(elapsed time).
–i	Print columns containing the I/O counts in the output.
–k	Instead of memory size, show total kcore-minutes.
–m	Show mean core size (the default).

−r	Show CPU factor (user time/(system-time + user-time).
−t	Show separate system and user CPU times.
−v	Exclude column headings from the output.
−l *line*	Show only processes belonging to terminal */dev/line.*
−u *user*	Show only processes belonging to *user* that may be specified by: a user ID, a login name that is then converted to a user ID, a **#**, which designates only those processes executed with superuser privileges, or **?**, which designates only those processes associated with unknown user IDs.
−g *group*	Show only processes belonging to *group*. The *group* may be designated by either the group ID or group name.
−s *time*	Select processes existing at or after *time*, given in the format *hr* [:*min* [:*sec*]].
−e *time*	Select processes existing at or before *time*.
−S *time*	Select processes starting at or after *time*.
−E *time*	Select processes ending at or before *time*. Using the same *time* for both **−S** and **−E** shows the processes that existed at *time*.
−n *pattern*	Show only commands matching *pattern* that may be a regular expression as in *ed*(1) except that + means one or more occurrences.
−q	Do not print any output records, just print the average statistics as with the **−a** option.
−o *ofile*	Copy selected process records in the input data format to *ofile*; suppress standard output printing.
−H *factor*	Show only processes that exceed *factor*, where factor is the "hog factor" as explained in option **−h** above.
−O *sec*	Show only processes with CPU system time exceeding *sec* seconds.
−C *sec*	Show only processes with total CPU time, system plus user, exceeding *sec* seconds.
−I *chars*	Show only processes transferring more characters than the cutoff number given by *chars*.

FILES

/etc/passwd
/usr/adm/pacct
/etc/group

SEE ALSO

ps(1), su(1).
acct(2), acct(4), utmp(4) in the *Programmer's Reference Manual.*
acct(1M), acctcms(1M), acctcon(1M), acctmerg(1M), acctprc(1M), acctsh(1M), fwtmp(1M), runacct(1M) in the *System Administrator's Reference Manual.*

BUGS

acctcom reports only on processes that have terminated; use *ps* for active processes. If *time* exceeds the present time, then *time* is interpreted as occurring on the previous day.

NAME

Add_disk – add a secondary disk to the system

SYNOPSIS

Add_disk [*disk number*]

DESCRIPTION

The *Add_disk* command allows the user to add an extra SCSI disk to the system.

The *disk number* option should be specified only if a disk number of 3 or more is needed as the default disk number is 2.

The Add_disk command creates the required directory, makes the appropriate device file links, makes a new filesystem, does the required mount operation, and adds the appropriate entry to */etc/fstab*.

NAME

admin – create and administer SCCS files

SYNOPSIS

admin [–**n**] [–**i**[name]] [–**r**rel] [–**t**[name]] [–**f**flag[flag-val]] [–**d**flag[flag-val]] [–**a**login] [–**e**login] [–**m**[mrlist]] [–**y**[comment]] [–**h**] [–**z**] files

DESCRIPTION

admin is used to create new SCCS files and change parameters of existing ones. Arguments to *admin*, which may appear in any order, consist of keyletter arguments, which begin with –, and named files (note that SCCS file names must begin with the characters s.). If a named file does not exist, it is created, and its parameters are initialized according to the specified keyletter arguments. Parameters not initialized by a keyletter argument are assigned a default value. If a named file does exist, parameters corresponding to specified keyletter arguments are changed, and other parameters are left as is.

If a directory is named, *admin* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with s.) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

OPTIONS

The keyletter arguments are as follows. Each is explained as though only one named file is to be processed since the effects of the arguments apply independently to each named file.

–**n**	This keyletter indicates that a new SCCS file is to be created.
–**i***[name]*	The *name* of a file from which the text for a new SCCS file is to be taken. The text constitutes the first delta of the file (see –**r** keyletter for delta numbering scheme). If the **i** keyletter is used, but the file name is omitted, the text is obtained by reading the standard input until an end-of-file is encountered. If this keyletter is omitted, then the SCCS file is created empty. Only one SCCS file may be created by an *admin* command on which the **i** keyletter is supplied. Using a single *admin* to create two or more SCCS files requires that they be created empty (no –**i** keyletter). Note that the –**i** keyletter implies the –**n** keyletter.

–r*rel* The *rel*ease into which the initial delta is inserted. This keyletter may be used only if the **–i** keyletter is also used. If the **–r** keyletter is not used, the initial delta is inserted into release 1. The level of the initial delta is always 1 (by default initial deltas are named 1.1).

–t*[name]* The *name* of a file from which descriptive text for the SCCS file is to be taken. If the **–t** keyletter is used and *admin* is creating a new SCCS file (the **–n** and/or **–i** keyletters also used), the descriptive text file name must also be supplied. In the case of existing SCCS files: (1) a **–t** keyletter without a file name causes removal of descriptive text (if any) currently in the SCCS file, and (2) a **–t** keyletter with a file name causes text (if any) in the named file to replace the descriptive text (if any) currently in the SCCS file.

–f*flag* This keyletter specifies a *flag*, and, possibly, a value for the *flag*, to be placed in the SCCS file. Several **f** keyletters may be supplied on a single *admin* command line. The allowable *flag*s and their values are:

> **b** Allows use of the **–b** keyletter on a *get*(1) command to create branch deltas.
>
> **c***ceil* The highest release (i.e., "ceiling"), a number greater than 0 but less than or equal to 9999, which may be retrieved by a *get*(1) command for editing. The default value for an unspecified **c** flag is 9999.
>
> **f***floor* The lowest release (i.e., "floor"), a number greater than 0 but less than 9999, which may be retrieved by a *get*(1) command for editing. The default value for an unspecified **f** flag is 1.
>
> **d***SID* The default delta number (SIDs+1) to be used by a *get*(1) command.
>
> **i***[str]* Causes the "No id keywords (ge6)" message issued by *get*(1) or *delta*(1) to be treated as a fatal error. In the absence of this flag, the message is only a warning. The message is issued if no SCCS identification keywords [see *get*(1)] are found in the text retrieved or stored in the SCCS file. If a value is supplied, the keywords must exactly

match the given string, however the string must contain a keyword, and no embedded newlines.

j Allows concurrent *get*(1) commands for editing on the same SIDs+1 of an SCCS file. This allows multiple concurrent updates to the same version of the SCCS file.

l*list* A *list* of releases to which deltas can no longer be made (**get −e** against one of these "locked" releases fails). The *list* has the following syntax:

<list> ::= <range> | <list> , <range>
<range>˜::= *RELEASE NUMBER* | **a**

The character **a** in the *list* is equivalent to specifying *all releases* for the named SCCS file.

n Causes *delta*(1) to create a "null" delta in each of those releases (if any) being skipped when a delta is made in a *new* release (e.g., in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). These null deltas serve as "anchor points" so that branch deltas may later be created from them. The absence of this flag causes skipped releases to be non-existent in the SCCS file, preventing branch deltas from being created from them in the future.

q*text* User definable text substituted for all occurrences of the %Q% keyword in SCCS file text retrieved by *get*(1).

m*mod* *Mod*ule name of the SCCS file substituted for all occurrences of the %M% keyword in SCCS file text retrieved by *get*(1). If the **m** flag is not specified, the value assigned is the name of the SCCS file with the leading **s.** removed.

t*type* *Type* of module in the SCCS file substituted for all occurrences of %Y% keyword in SCCS file text retrieved by *get*(1).

v*pgm* Causes *delta*(1) to prompt for Modification Request (MR) numbers as the reason for creating a delta. The optional value specifies the name of an MR number validity checking program [see *delta*(1)]. (If this flag is set when creating an SCCS file, the **m** keyletter must also be used even if its value is null).

−d*flag*	Causes removal (deletion) of the specified *flag* from an SCCS file. The −**d** keyletter may be specified only when processing existing SCCS files. Several −**d** keyletters may be supplied on a single *admin* command. See the −**f** keyletter for allowable *flag* names.
l*list*	A *list* of releases to be "unlocked". See the −**f** keyletter for a description of the **l** flag and the syntax of a *list*.
−**a***login*	A *login* name, or numerical UNIX system group ID, to be added to the list of users which may make deltas (changes) to the SCCS file. A group ID is equivalent to specifying all *login* names common to that group ID. Several **a** keyletters may be used on a single *admin* command line. As many *login*s, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty, then anyone may add deltas. If *login* or group ID is preceded by a **!** they are to be denied permission to make deltas.
−**e***login*	A *login* name, or numerical group ID, to be erased from the list of users allowed to make deltas (changes) to the SCCS file. Specifying a group ID is equivalent to specifying all *login* names common to that group ID. Several **e** keyletters may be used on a single *admin* command line.
−**m***[mrlist]*	The list of Modification Requests (MR) numbers is inserted into the SCCS file as the reason for creating the initial delta in a manner identical to *delta*(1). The **v** flag must be set and the MR numbers are validated if the **v** flag has a value (the name of an MR number validation program). Diagnostics will occur if the **v** flag is not set or MR validation fails.
−**y***[comment]*	The *comment* text is inserted into the SCCS file as a comment for the initial delta in a manner identical to that of *delta*(1). Omission of the −**y** keyletter results in a default comment line being inserted in the form: date and time created *YY/MM/DD HH:MM:SS* by *login*

The −y keyletter is valid only if the **−i** and/or **−n** keyletters are specified (i.e., a new SCCS file is being created).

−h Causes *admin* to check the structure of the SCCS file [see *sccsfile*(5)], and to compare a newly computed check-sum (the sum of all the characters in the SCCS file except those in the first line) with the check-sum that is stored in the first line of the SCCS file. Appropriate error diagnostics are produced. keyletter inhibits writing on the file, so that it nullifies the effect of any other keyletters supplied, and is, therefore, only meaningful when processing existing files.

−z The SCCS file check-sum is recomputed and stored in the first line of the SCCS file (see **−h**, above).

Note that use of this keyletter on a truly corrupted file may prevent future detection of the corruption.

The last component of all SCCS file names must be of the form **s.***file-name*. New SCCS files are given mode 444 [see *chmod*(1)]. Write permission in the pertinent directory is, of course, required to create a file. All writing done by *admin* is to a temporary x-file, called **x.***file-name*, [see *get*(1)], created with mode 444 if the *admin* command is creating a new SCCS file, or with the same mode as the SCCS file if it exists. After successful execution of *admin*, the SCCS file is removed (if it exists), and the x-file is renamed with the name of the SCCS file. This ensures that changes are made to the SCCS file only if no errors occurred.

It is recommended that directories containing SCCS files be mode 755 and that SCCS files themselves be mode 444. The mode of the directories allows only the owner to modify SCCS files contained in the directories. The mode of the SCCS files prevents any modification at all except by SCCS commands.

If it should be necessary to patch an SCCS file for any reason, the mode may be changed to 644 by the owner allowing use of *ed*(1). *Care must be taken!* The edited file should *always* be processed by an **admin −h** to check for corruption followed by an **admin −z** to generate a proper check-sum. Another **admin −h** is recommended to ensure the SCCS file is valid.

admin also makes use of a transient lock file (called z.*file-name*), which is used to prevent simultaneous updates to the SCCS file by different users. See *get*(1) for further information.

FILES

g-file	Existed before the execution of *delta*; removed after completion of *delta*.
p-file	Existed before the execution of *delta*; may exist after completion of *delta*.
q-file	Created during the execution of *delta*; removed after completion of *delta*.
x-file	Created during the execution of *delta*; renamed to SCCS file after completion of *delta*.
z-file	Created during the execution of *delta*; removed during the execution of *delta*.
d-file	Created during the execution of *delta*; removed after completion of *delta*.
/usr/bin/bdiff	Program to compute differences between the ``gotten'' file and the *g-file*.

SEE ALSO

delta(1), get(1), prs(1), what(1), sccsfile(4).
ed(1), help(1) in the *User's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

ansitape – ANSI standard tape handler

SYNOPSIS

ansitape [key] [keyargs] [files]

DESCRIPTION

Ansitape reads and writes magnetic tapes written in ANSI standard format (called "Files-11" by DEC). Tapes written by *ansitape* are labeled with the first 6 characters of the machine name by default. Actions are controlled by the *key* argument. The *key* is a string of characters containing at most one function letter. Other arguments to the command are a tape label and file names specifying which files are to be written onto or extracted from the tape.

The function portion of the key is specified by one of the following letters:

r The named files are written at the end of the tape. The **c** function implies this.

x The named files are extracted from the tape. If no file argument is given, the entire contents of the tape is extracted. Note that if the tape has duplicated file names, only the last file of a given name can be extracted.

t The names of the specified files are listed each time they occur on the tape. If no file argument is given, all files on the tape are listed.

c Create a new tape; writing begins at the beginning of the tape instead of after the last file. This command implies **r**.

The following characters may be used in addition to the letter which selects the function desired.

f This argument allows the selection of a different tape device. The next word in the keyargs list is taken to be the full name of a device to write the tape on. The default is /dev/tape.

n The **n** option allows the user to specify as the next argument in the keyargs list, a control file containing the names of files to put on the tape. If the file name is '–', the control file will, instead, be read from standard input. The control file contains one line for each file to be placed on the tape. Each line has two names, the name of the file on the local machine, and the name it is to have when placed on the tape. This allows for more convenient flattening of hierarchies when placing them on tape. If the second name is omitted, the UNIX file name will be used on the tape also. This argument can only be used with the **r** and **c** functions.

l The **l** option allows the user to specify the label to be placed on the tape. The next argument in the keyargs list is taken as the tape label, which will be space padded or truncated to six characters. This option is meaningless unless **c** is also specified.

v Normally *ansitape* works relatively silently. The v (verbose) option causes it to type information about each file as it processes it.

b The **b** option allows the user to select the blocksize to be used for the tape. By default, *ansitape* uses the maximum block size permitted by the ANSI standard, 2048. Some systems will permit a much larger block size, and if large files are being put on the tape it may be advantageous to do so. *Ansitape* will take the next argument of the keyargs list as the blocksize for the tape. Values below 18 or above 32k will be limited to that range. The standard scale factors b=512 and k=1024 are accepted.

F The **F** flag allows *ansitape* to write ansi 'D' format fixed record length tapes. The next two keyargs must be the recordsize and blocksize to be used, with the same scale factors and range limits as for the **b** option. The files to be written by the **F** flag must be in fixed format on the unix end — all lines should be *EXACTLY* **recordsize** bytes long plus a terminating newline (which will be discarded). Note that this is exactly the same format produced by *ansitape* when reading an ansi 'D' format tape.

Ansitape will not copy directories, character or block special files, symbolic links, sockets, or binary executables. Attempts to put these on tape will result in warnings, and they will be skipped completely.

FILES

/dev/tape default tape drive

DIAGNOSTICS

A warning message will be generated when a record exceeds the maximum record length and the affected file will be truncated.

BUGS

Ansitape quietly truncates names longer than 17 characters.

Multivolume tapes can be read (provided no files cross the volume boundary) but not written.

SEE ALSO

vmsprep(1), mtio(7)

NAME

apply – apply a command to a set of arguments

SYNOPSIS

apply [–ac] [–n] command args ...

DESCRIPTION

Apply runs the named *command* on each argument *arg* in turn. Normally arguments are chosen singly; the optional number *n* specifies the number of arguments to be passed to *command.* If *n* is zero, *command* is run without arguments once for each *arg*. Character sequences of the form *%d* in *command,* where *d* is a digit from 1 to 9, are replaced by the *d*'th following unused *arg*. If any such sequences occur, *n* is ignored, and the number of arguments passed to *command* is the maximum value of *d* in *command.* The character '%' may be changed by the –a option.

EXAMPLES

```
        apply echo *
```
is similar to *ls*(1);
```
        apply –2 cmp a1 b1 a2 b2 ...
```
compares the 'a' files to the 'b' files;
```
        apply –0 who 1 2 3 4 5
```
runs *who*(1) 5 times; and
```
        apply ´ln %1 /usr/joe´ *
```
links all files in the current directory to the directory /usr/joe.

SEE ALSO

sh(1)

BUGS

Shell metacharacters in *command* may have bizarre effects; it is best to enclose complicated commands in single quotes ´ ´.

There is no way to pass a literal '%2' if '%' is the argument expansion character.

NAME

appres - list application resource database

SYNOPSIS

appres [[classname [instancename]] [-xrm resource ...]

DESCRIPTION

The *appres* program prints the resources seen by an application of the specified *class* and *instance* name. It is used to determine which resources a particular program would load. For example,

```
% appres XTerm
```

would list the resources that any *xterm* program would load. To also match particular instance names,

```
% appres myxterm XTerm
```

If no application class is specified, the class *-NoSuchClass-* (which should have no defaults) is used.

SEE ALSO

X(1), xrdb(1), listres(1)

COPYRIGHT

Copyright 1989, Massachusetts Institute of Technology.
See *X(1)* for a full statement of rights and permissions.

AUTHOR

Jim Fulton, MIT X Consortium

NAME

apropos – locate commands by keyword lookup

SYNOPSIS

apropos keyword ...

DESCRIPTION

apropos shows which manual sections contain instances of any of the given keywords in their title. Each word is considered separately and case of letters is ignored. Words which are part of other words are considered; thus, when looking for compile, *apropos* will find all instances of 'compiler' also. Try

```
apropos password
```

and

```
apropos editor
```

If the line starts 'name(section) ...' you can do 'man section name' to get the documentation for it. Try 'apropos format' and then 'man 3s printf' to get the manual on the subroutine *printf*.

apropos is actually just the –k option to the *man*(1) command.

FILES

/usr/catman/whatis	data base

SEE ALSO

man(1), whatis(1), makewhatis(1M)

NAME

ar – archive and library maintainer

SYNOPSIS

ar option [*posname*] file1 ...

DESCRIPTION

The archiver (**ar**) maintains groups of files as a single archive file. Generally, you use this utility to create and update library files that the link editor uses; however, you can use the archiver for any similar purpose.

Any option that changes an object library causes the symbol table to be updated. This makes adding one file at a time to a library very slow.

NOTE: This version uses a portable ASCII-format archive that you can use on various machines that run UNIX.

file1 in the synopsis is the "archive file" referred to below.

In the text, option refers to a character (from the set **drqtpmx**) that you can concatenate with one or more of **svuaibclo.** A suboption refers to options (from the set **abiou**) that you can only use with other options.

The options do these things:

d Deletes the specified files from the archive file.

r Replaces the specified files in the archive file. If you use the suboption **u** with **r,** the archiver only replaces those files that have 'last-modified' dates later than the archive files. If you use a positioning character (from the set **abi**) you must specify the *posname* argument to tell the archiver to put the new files after (**a**) or before (**b** or **i**). Otherwise, the archiver puts new files at the end of the archive.

q Appends the specified files to the end of the archive file. The archiver does not accept suboption positioning characters with the **q** option. It also does not check whether the files you want to add already exist in the archive. Since the symbol table of an object library is updated with **q** it is advisable to add as many files as possible in one execution of *ar*. One cannot avoid quadratic behavior when creating a large archive piece by piece.

t Prints a table of contents for the files in the archive file. If you don't specify any file names, the archiver builds a table of contents for all files. If you specify file names, the archiver builds a table of contents only for those files.

p Prints the specified files from the archive.

m Moves the specified files to the end of the archive. If you specify a positioning character, you must also specify the *posname* (as in option **r**) to tell the archiver where to move the files.

x Extracts the specified files from the archive. If you don't specify any file names, the archiver extracts all files. When it extracts files, the archiver does not change any file. Normally, the 'last-modified' date for each extracted file shows the date when someone extracted it; however, when you use **o**, the archiver resets the 'last-modified' date to the date recorded in the archive.

s Makes a symbol definition (symdef file) as the first file of an archive. This file contains a hash table of *ranlib* structures and a corresponding string table. The symdef file's name is based on the byte ordering of the hash table and the byte ordering of the file's target machine. Files must be consistent in their target byte ordering before the archiver can create a symdef file. The s option is automatically added when any of the options **d**, **m**, or **r** is requested. If you specify **s**, the archiver creates the symdef file as its last action before finishing execution. You must specify at least one other archive option (**m**, **p**, **d**, **r**, or **t**) when you use the s option.

v Gives a verbose file-by-file description as the archiver makes a new archive file from an old archive and its constituent files. When you use this option with **t**, the archiver lists all information about the files in the archive. When you use this option with **p**, the archiver precedes each file with a name.

c Suppresses the normal message that the archiver prints when it creates the specified archive file. Normally, the archiver creates the specified archiver file when it needs to.

l Puts temporary files in the local directory. If option **l** is not supplied and the environment variable **TMPDIR** is defined then **TMPDIR**'s value is used as the name of the directory for temporary files. If neither option **l** nor **TMPDIR** is supplied, the archiver puts its temporary files in the directory */tmp*.

The suboptions do these things:

a Specifies that the file goes after the existing file (*posname*). Use this suboption with the **m** or **r** options.

b	Specifies that the file goes before the existing file (*posname*). Use this suboption with the **m** or **r** options.
i	Specifies that the file goes before the existing file (*posname*). Use this suboption with the **m** or **r** options.
o	Forces a newly created file to have the 'last modified' date that it had before it was extracted from the archive. Use this suboption with the **x** option.
u	Prevents the archiver from replacing an existing file unless the replacement is newer than the existing file. This option uses the UNIX system 'last modified' data for this comparison. Use this suboption with the **r** option. **u** gives no warning when replacement is refused.

FILES

/tmp/v* or TMPDIR/v* temporaries

SEE ALSO

lorder(1), ld(1), odump(1), ranhash(3X), ar(4)

NOTES

There is no *ranlib* program in IRIX. Option **s** creates the symbol table *ld* uses.

Options **r**, **d**, **m**, and **q** imply option **s**. Since option **s** creates a symbol table, creating an object library by executing *ar* once per object file will be very slow. Creating an object library with a single execution of *ar* is much faster.

DIAGNOSTICS

xxxxx not found

The file **xxxxx** was not found in the archive. It could mean a simple misspelling, but it could also mean that you supplied **xxxxx** more than the number of times **xxxxx** appears in the archive! Files not found change the exit code from *ar* but any attempted update of the archive (by option **r** for example) is not suppressed.

d123456789abcdefghi not found. d123456789abcde is in the archive, however

The file you wanted was not found (d123... here is just an example of course). However, there was a file in the archive whose name, if truncated to 15 characters, would be found. This message is a hint that the name may have been truncated to fit in the *ar* format. Try *ar* again using the shorter name. Files not found change the exit code from *ar* but any attempted update of the archive (by option **r** for example) is not suppressed.

not in archive format

You probably forgot to specify the archive name in the command. The *file1* mentioned in the synopsis should be the archive name.

MORE NOTES

The behavior documented in this section is not guaranteed to remain the same across releases. This section is provided as help in case *ar* does something surprising.

If there is only one hard link (ie, at most one non-symbolic-link. see *ln*(2)) to an archive which is being updated then an old archive contents are replaced by the new contents by *rename*(2). Otherwise, when updating, replacement is by copying the new data onto the old file. If the archive is updated, the replacement archive is built in the same directory as the named archive (after following symbolic links to the location of the named archive).

In case the copy operation mentioned above is interrupted in mid-copy (which is normally not possible) *ar* will attempt to set the archive length to 0 and the modification-date to January 1, 1970 as a hint that the archive is not usable.

If the *ar* command results in an unchanged archive, the old archive will not be replaced. This is best achieved with, for example, *ar ru lib.a x.o*; if the named object file is not put into the archive, the archive is not modified. The definition of unchanged is very conservative: *ar r lib.a x.o*, for example, always changes the archive since x.o is added or replaced (even though x.o itself may be unchanged).

The following is a sampling of traditional *ar* behaviors that you may find surprising.

If you specify the same file twice in an argument list, it can appear twice in the archive file.

The o option does not change the 'last-modified' date of a file unless you own the extracted file or you are the super-user.

File names longer than 15 characters are truncated to 15 characters since the format does not allow longer file names. Since *ar* files are defined as a portable format we must live with the name size limit.

Only the final component of a path-name is recorded in an archive. Example:

```
ar cr x.a /a/b/c/d123456789abcdefghij///
ar  x x.a /a/b/c/d123456789abcdefghij///
```

Trailing slashes and leading directories in the path are stripped from the name recorded in the archive. The `ar cr` above will show file ''d123456789abcde'' in the archive (assuming ''d123456789abcdefghij'' exists at the named location when *ar* is run). The `ar x` above will not find a file, since the name given does not match any name in the archive (the name given is longer than the name stored in the archive).

If you specify two names longer than 15 characters (in an `ar cr` or the like) which match in the first 15 characters both will be truncated to 15 characters and both will be put in the archive.

If *ar* truncates a name, it prints a message to that effect on standard error.

If you give *ar* the same name twice in an `ar x` command the second instance of the name will provoke a ''not found'' message.

NAME

as – MIPS assembler

SYNOPSIS

as [option] ... file ...

DESCRIPTION

as is the MIPS assembler. It produces files in the following formats: MIPS object code in MIPS extended *coff* format (the normal result) and binary assembly language. *As* never runs the loader. *As* accepts several types of arguments:

The argument *file* is assumed to be symbolic assembly language source program. It is assembled, producing an object file.

as always defines the preprocessor symbols LANGUAGE_ASSEMBLY, _LANGUAGE_ASSEMBLY, , sgi, host_mips, mips, unix, SVR3, SYSTYPE_SYSV, _SYSTYPE_SYSV, _MIPSEB, and MIPSEB. These are shown by the –v option to *as*(1). The C preprocessor adds other standard definitions of its own (see *cpp*(1)).

The following options are interpreted by *as* and have the same meaning in *cc*(1).

–g0 Have the assembler produce no symbol table information for symbolic debugging. This is the default.

–g1 Have the assembler produce additional symbol table information for accurate but limited symbolic debugging of partially optimized code.

–g or **–g2**
Have the assembler produce additional symbol table information for full symbolic debugging and not do optimizations that limit full symbolic debugging.

–g3 Have the assembler produce additional symbol table information for full symbolic debugging for fully optimized code. This option makes the debugger inaccurate.

–w Suppress warning messages.

–P Run only the C macro preprocessor and put the result for each '.s' file in a corresponding '.i' file. The '.i' file has no '#' lines in it.

–E Run only the C macro preprocessor on the specified assembly source files, and send the result to the standard output.

−o *output*

Name the final output file *output*. If this option is used, the file 'a.out' is undisturbed.

−D*name*=*def*

−D*name*

Define the *name* to the C macro preprocessor, as if by '#define'. If no definition is given, the name is defined as "1".

−U*name*

Remove any initial definition of *name*.

−I*dir* '#include' files whose names do not begin with '/' are always sought first in the directory of the *file* argument, then in directories specified in −I options, and finally in the standard directory (**/usr/include**).

−I This option will cause '#include' files never to be searched for in the standard directory (**/usr/include**).

−G *num*

Specify the maximum size, in bytes, of a data item that is to be accessed from the global pointer. *Num* is assumed to be a decimal number. If *num* is zero, no data is accessed from the global pointer. The default value for *num* is 8 bytes. Please note that this switch is non-standard and may not be supported across product lines.

−v Print the passes as they execute with their arguments and their input and output files.

−V Print the version of the driver and the versions of all passes. This is done with the *what*(1) command. Please note that this switch is non-standard and may not be supported across product lines.

−cpp Run the C macro preprocessor on assembly source files before compiling. This is the default for *as*(1).

−nocpp Do not run the C macro preprocessor on assembly source before compiling.

−m Apply the M4 preprocessor to the source file before assembling it.

The options described below primarily aid compiler development and are not generally used:

−H*c* Halt compiling after the pass specified by the character *c*, producing an intermediate file for the next pass. The *c* can be [a]. It selects the assembler pass in the same way as the −t option. If this option is used, the symbol table file produced and used by the passes, is the last component of the source file with the suffix changed to '.T' and is not removed. Please note that this switch is non-standard and may not be supported across product lines.

−K Build and use intermediate file names with the last component of the source file's name replacing its suffix with the conventional suffix for the type of file (for example '.G' file for binary assembly language) These intermediate files are never removed even when a pass encounters a fatal error. Please note that this switch is non-standard and may not be supported across product lines.

−W*c,arg1[,arg2...]*
Pass the argument[s] *argi* to the assembler pass *c*. The *c* is one of [**pab**]. The c selects the assembler pass in the same way as the **−t** option.

The options **−t[hpab]**, **−h***path*, and **−B***string* select a name to use for a particular pass. These arguments are processed from left to right so their order is significant. When the **−B** option is encountered, the selection of names takes place using the last **−h** and **−t** options. Therefore, the **−B** option is always required when using **−h** or **−t**. Sets of these options can be used to select any combination of names.

−t[hpab]
Select the names. The names selected are those designated by the characters following the **−t** option according to the following table:

Name	Character
include	h
cpp	p
as0	a
as1	b

If the character 'h' is in the **−t** argument then a directory is added to the list of directories to be used in searching for '#include' files. This directory name has the form */usr/*include*string*. This directory is to contain the include files for the *string* release of the compiler. The standard directory is still searched.

−h*path* Use *path* rather than the directory where the name is normally found. Please note that this switch is non-standard and may not be supported across product lines.

−B*string*
Append *string* to all names specified by the **−t** option. If no **−t** option has been processed before the **−B**, the **−t** option is assumed to be "hpab". This list designates all names.

If the environment variable TMPDIR is set, the value is used as the directory to place any temporary files rather than the default **/tmp**.

Invoking the assembler with a name of the form as*string* has the same effect as using a –**B***string* option on the command line.

Other arguments are ignored.

FILES

file.o	object file
a.out	assembler output
/tmp/ctm*	temporaries
/usr/lib/cpp	C macro preprocessor
/usr/lib/as0	symbolic to binary assembly language translator
/usr/lib/as1	binary assembly language assembler and reorganizer
/usr/include	standard directory for '#include' files

BUGS

The assembler attempts to continue after finding semantic errors. These errors may result in internal errors.

SEE ALSO

MIPS Assembly Language Programmer's Guide
cc(1), what(1)

DIAGNOSTICS

The diagnostics produced by the assembler are intended to be self-explanatory.

NOTES:

The environment variables COMP_TARGET_ROOT, TOOLROOT, RLS_ID_OBJECT are used by *as* while compiling the compiler itself. Their meaning is left undefined intentionally. They should **not** be set in your environment. ˜

NAME

at, batch – execute commands at a later time

SYNOPSIS

at [–q queue] time [date] [+increment]
at *–r* job...
at *–l* [*job* ...]
batch

DESCRIPTION

at and *batch* read commands from standard input to be executed at a later time. *at* allows you to specify when the commands should be executed, while jobs queued with *batch* will execute when system load level permits. *at* may be used with the following options:

–r	Removes jobs previously scheduled with *at*.
–l	Reports all jobs scheduled for the invoking user.
–q	Put job onto specified queue.

Standard output and standard error output are mailed to the user unless they are redirected elsewhere. The shell environment variables, current directory, umask, and ulimit are retained when the commands are executed. Open file descriptors, traps, and priority are lost.

Users are permitted to use *at* if their name appears in the file **/usr/lib/cron/at.allow**. If that file does not exist, the file **/usr/lib/cron/at.deny** is checked to determine if the user should be denied access to *at*. If neither file exists, only root is allowed to submit a job. If **at.deny** is empty, global usage is permitted. The allow/deny files consist of one user name per line. These files can only be modified by the superuser.

All jobs are placed on a specific queue. By default, *at* jobs go on queue "a", *batch* jobs go on queue "b". Additional queues may be created by altering the **/usr/lib/cron/queuedefs** file. *queuedefs*(4) defines the format for that file.

The *time* may be specified as 1, 2, or 4 digits. One and two digit numbers are taken to be hours, four digits to be hours and minutes. The time may alternately be specified as two numbers separated by a colon, meaning *hour*:*minute*. A suffix **am** or **pm** may be appended; otherwise a 24-hour clock time is understood. The suffix **zulu** may be used to indicate GMT. The special names **noon**, **midnight**, **now**, and **next** are also recognized.

An optional *date* may be specified as either a month name followed by a day number (and possibly year number preceded by an optional comma) or a day of the week (fully spelled or abbreviated to three characters). Two special "days", **today** and **tomorrow** are recognized. If no *date* is given, **today** is assumed if the given hour is greater than the current hour and

tomorrow is assumed if it is less. If the given month is less than the current month (and no year is given), next year is assumed.

The optional *increment* is simply a number suffixed by one of the following: **minutes, hours, days, weeks, months,** or **years.** (The singular form is also accepted.)

Thus legitimate commands include:

```
at 0815am Jan 24
at 8:15am Jan 24
at now + 1 day
at 5 pm Friday
```

at and *batch* write the job number and schedule time to standard error.

batch submits a batch job. It is almost equivalent to "at now", but not quite. For one, it goes into a different queue. For another, "at now" will respond with the error message **too late.**

at **-r** removes jobs previously scheduled by *at* or *batch*. The job number is the number given to you previously by the *at* or *batch* command. You can also get job numbers by typing *at* **-l**. You can only remove your own jobs unless you are the super-user.

EXAMPLES

The *at* and *batch* commands read from standard input the commands to be executed at a later time. *sh*(1) provides different ways of specifying standard input. Within your commands, it may be useful to redirect standard output.

This sequence can be used at a terminal:

```
batch
sort filename >outfile
<control-D> (hold down 'control' and depress 'D')
```

This sequence, which demonstrates redirecting standard error to a pipe, is useful in a shell procedure (the sequence of output redirection specifications is significant):

```
batch <<!
sort filename 2>&1 >outfile | mail loginid
!
```

To have a job reschedule itself, invoke *at* from within the shell procedure, by including code similar to the following within the shell file:

```
echo "sh shellfile" | at 1900 thursday next week
```

FILES

/usr/lib/cron	main cron directory
/usr/lib/cron/at.allow	list of allowed users
/usr/lib/cron/at.deny	list of denied users
/usr/lib/cron/queue	scheduling information
/usr/spool/cron/atjobs	spool area
/usr/lib/cron/.proto	generic prototype, prepended to all jobs
/usr/lib/cron/.proto.x	prototype for queue 'x'
/usr/lib/cron/queuedefs	definitions for queues

SEE ALSO

kill(1), mail(1), nice(1), ps(1), sh(1), sort(1).
cron(1M) in the *System Administrator's Reference Manual.*
queuedefs(4), proto(4) in the *Programmer's Reference Manual.*

DIAGNOSTICS

Complains about various syntax errors and times out of range.

NAME

awk – pattern scanning and processing language

SYNOPSIS

awk [–Fc] [prog] [parameters] [files]

DESCRIPTION

awk scans each input *file* for lines that match any of a set of patterns specified in *prog*. With each pattern in *prog* there can be an associated action that will be performed when a line of a *file* matches the pattern. The set of patterns may appear literally as *prog*, or in a file specified as **–f** *file*. The *prog* string should be enclosed in single quotes (′) to protect it from the shell.

Parameters, in the form x=... y=... etc., may be passed to *awk.*

Files are read in order; if there are no files, the standard input is read. The file name – means the standard input. Each line is matched against the pattern portion of every pattern-action statement; the associated action is performed for each matched pattern.

An input line is made up of fields separated by white space. (This default can be changed by using FS; see below). The fields are denoted **$1**, **$2**, . . . ; **$0** refers to the entire line.

A pattern-action statement has the form:

```
pattern { action }
```

A missing action means print the line; a missing pattern always matches. An action is a sequence of statements. A statement can be one of the following:

```
if ( conditional ) statement [ else statement ]
while ( conditional ) statement
for ( expression ; conditional ; expression ) statement
break
continue
{ [ statement ] ... }
variable = expression
print [ expression-list ] [ >expression ]
printf format [ , expression-list ] [ >expression ]
next      # skip remaining patterns on this input line
exit      # skip the rest of the input
```

Statements are terminated by semicolons, new-lines, or right braces. An empty expression-list stands for the whole line. Expressions take on string or numeric values as appropriate, and are built using the operators +, –, *, /, %, and concatenation (indicated by a blank). The C operators ++, —, +=, –=, *=, /=, and %= are also available in expressions. Variables may be scalars, array elements (denoted x[i]) or fields. Variables are initialized to the null string. Array subscripts may be any string, not necessarily numeric; this allows for a form of associative memory. String constants are quoted (").

The *print* statement prints its arguments on the standard output (or on a file if >*expr* is present), separated by the current output field separator, and terminated by the output record separator. The *printf* statement formats its expression list according to the format [see *printf*(3S) in the *Programmer's Reference Manual*].

The built-in function *length* returns the length of its argument taken as a string, or of the whole line if no argument. There are also built-in functions *exp*, *log*, *sqrt*, and *int*. The last truncates its argument to an integer; *substr*(*s*, *m*, *n*) returns the *n*-character substring of *s* that begins at position *m*. The function *sprintf*(*fmt*, *expr*, *expr*, ...) formats the expressions according to the *printf*(3S) format given by *fmt* and returns the resulting string.

Patterns are arbitrary Boolean combinations (!, | | , &&, and parentheses) of regular expressions and relational expressions. Regular expressions must be surrounded by slashes and are as in *egrep* (see *grep*(1)). Isolated regular expressions in a pattern apply to the entire line. Regular expressions may also occur in relational expressions. A pattern may consist of two patterns separated by a comma; in this case, the action is performed for all lines between an occurrence of the first pattern and the next occurrence of the second.

A relational expression is one of the following:

```
expression matchop regular-expression
expression relop expression
```

where a relop is any of the six relational operators in C, and a matchop is either ~ (for *contains*) or !~ (for *does not contain*). A conditional is an arithmetic expression, a relational expression, or a Boolean combination of these.

The special patterns BEGIN and END may be used to capture control before the first input line is read and after the last. BEGIN must be the first pattern, END the last.

A single character *c* may be used to separate the fields by starting the program with:

```
BEGIN { FS = c }
```

or by using the −F*c* option.

Other variable names with special meanings include NF, the number of fields in the current record; NR, the ordinal number of the current record; FILENAME, the name of the current input file; OFS, the output field separator (default blank); ORS, the output record separator (default new-line); and OFMT, the output format for numbers (default **%.6g**).

EXAMPLES

Print lines longer than 72 characters:

```
length > 72
```

Print first two fields in opposite order:

```
{ print $2, $1 }
```

Add up first column, print sum and average:

```
        { s += $1 }
END     { print "sum is", s, " average is", s/NR }
```

Print fields in reverse order:

```
{ for (i = NF; i > 0; −−i) print $i }
```

Print all lines between start/stop pairs:

```
/start/, /stop/
```

Print all lines whose first field is different from previous one:

```
$1 != prev { print; prev = $1 }
```

Print file, filling in page numbers starting at 5:

```
/Page/ { $2 = n++; }
       { print }
```

command line: awk −f program n=5 input

SEE ALSO

grep(1), sed(1), nawk(1)
lex(1), printf(3S) in the *Programmer's Reference Manual.*

BUGS

Input white space is not preserved on output if fields are involved.
There are no explicit conversions between numbers and strings. To force an expression to be treated as a number add 0 to it; to force it to be treated as a string concatenate the null string ("") to it.

NAME

Backup – backup the specified file or directory

SYNOPSIS

Backup [*–h hostname*] [*directory name* | *file name*]

DESCRIPTION

The *Backup* command archives the named file or directory ("." if none specified) to the local or remote tape device. It can be used to make a full system backup by specifying the directory name as /.

In case of a full backup this command makes a list of the files in the disk volume header and saves this information in a file which is then stored on tape. This file is used during crash recovery to restore a damaged volume header. The current date is saved in the file **/etc/lastbackup.**

If a tape drive attached to a remote host is used for backup, the name of the remote host needs to be specified with the **–h hostname** option on the command line. For remote backup to successfully work, the user should have a TCP/IP network connection to the remote host and also have "guest" login privileges on that host.

The Backup command uses **bru** to perform the backup function.

FILES

/tmp/volhdrlist	– contains the list of the root volume header files
/etc/lastbackup	– contains the date of last full backup

SEE ALSO

Restore(1), List_tape(1), bru(1).

NAME

banner – make posters

SYNOPSIS

banner strings

DESCRIPTION

banner prints its arguments (each up to 10 characters long) in large letters on the standard output.

SEE ALSO

echo(1).

NAME

basename, dirname – deliver portions of path names

SYNOPSIS

basename string [suffix]
dirname string

DESCRIPTION

basename deletes any prefix ending in / and the *suffix* (if present in *string*) from *string*, and prints the result on the standard output. It is normally used inside substitution marks (\` \`) within shell procedures.

Dirname delivers all but the last level of the path name in *string*.

EXAMPLES

The following example, invoked with the argument **/usr/src/cmd/cat.c**, compiles the named file and moves the output to a file named *cat* in the current directory:

```
cc $1
mv a.out `basename $1 '\.c'`
```

The following example will set the shell variable NAME to **/usr/src/cmd**:

```
NAME=`dirname /usr/src/cmd/cat.c`
```

SEE ALSO

sh(1).

NAME

bc – arbitrary-precision arithmetic language

SYNOPSIS

bc [**–c**] [**–l**] [file ...]

DESCRIPTION

bc is an interactive processor for a language that resembles C but provides unlimited precision arithmetic. It takes input from any files given, then reads the standard input. The *bc*(1) utility is actually a preprocessor for *dc*(1), which it invokes automatically unless the **–c** option is present. In this case the *dc* input is sent to the standard output instead. The options are as follows:

–c Compile only. The output is send to the standard output.

–l Argument stands for the name of an arbitrary precision math library.

The syntax for *bc* programs is as follows; L means letter a–z, E means expression, S means statement.

Comments
  are enclosed in /* and */.

Names
  simple variables: L
  array elements: L [E]
  The words ''ibase'', ''obase'', and ''scale''

Other operands
  arbitrarily long numbers with optional sign and decimal point.
  (E)
  sqrt (E)
  length (E) number of significant decimal digits
  scale (E) number of digits right of decimal point
  L (E , ... , E)

Operators
  + – * / % ^ (% is remainder; ^ is power)
  ++ — (prefix and postfix; apply to names)
  == <= >= != < >
  = =+ =– =* =/ =% =^

Statements
  E
  { S ; ... ; S }
  if (E) S
  while (E) S
  for (E ; E ; E) S

```
        null statement
        break
        quit
Function definitions
        define L ( L ,..., L ) {
                auto L, ... , L
                S; ... S
                return ( E )
        }
Functions in –l math library
        s(x)    sine
        c(x)    cosine
        e(x)    exponential
        l(x)    log
        a(x)    arctangent
        j(n,x)  Bessel function
```

All function arguments are passed by value.

The value of a statement that is an expression is printed unless the main operator is an assignment. Either semicolons or new-lines may separate statements. Assignment to *scale* influences the number of digits to be retained on arithmetic operations in the manner of *dc*(1). Assignments to *ibase* or *obase* set the input and output number radix respectively.

The same letter may be used as an array, a function, and a simple variable simultaneously. All variables are global to the program. ''Auto'' variables are pushed down during function calls. When using arrays as function arguments or defining them as automatic variables, empty square brackets must follow the array name.

EXAMPLE

```
scale = 20
define e(x){
        auto a, b, c, i, s
        a = 1
        b = 1
        s = 1
        for(i=1; 1==1; i++){
                a = a*x
                b = b*i
                c = a/b
                if(c == 0) return(s)
                s = s+c
        }
}
```

defines a function to compute an approximate value of the exponential function and

```
        for(i=1; i<=10; i++) e(i)
```

prints approximate values of the exponential function of the first ten integers.

FILES

/usr/lib/lib.b	mathematical library
/usr/bin/dc	desk calculator proper

SEE ALSO

dc(1).

BUGS

The *bc* command does not yet recognize the logical operators, && and | |.
For statement must have all three expressions (E's).
Quit is interpreted when read, not when executed.

NAME

bdftosnf - BDF to SNF font compiler for X11

SYNOPSIS

bdftosnf [-p#] [-u#] [-m] [-l] [-M] [-L] [-w] [-W] [-t] [-i] [bdf-file]

DESCRIPTION

bdftosnf reads a Bitmap Distribution Format (BDF) font from the specified file (or from standard input if no file is specified) and writes an X11 server normal font (SNF) to standard output.

OPTIONS

−p#	Force the glyph padding to a specific number. The legal values are 1, 2, 4, and 8. **−u#** Force the scanline unit padding to a specific number. The legal values are 1, 2, and 4.
−m	Force the bit order to most significant bit first.
−l	Force the bit order to least significant bit first.
−M	Force the byte order to most significant bit first.
−L	Force the byte order to least significant bit first.
−w	Print warnings if the character bitmaps have bits set to one outside of their defined widths.
−W	Print warnings for characters with an encoding of -1; the default is to silently ignore such characters.
−t	Expand glyphs in "terminal-emulator" fonts to fill the bounding box.
−i	Don't compute correct ink metrics for "terminal-emulator" fonts.

SEE ALSO

X(1), Xserver(1)
"Bitmap Distribution Format 2.1"

NAME

bfs – big file scanner

SYNOPSIS

bfs [–] name

DESCRIPTION

The *bfs* command is (almost) like *ed*(1) except that it is read-only and processes much larger files. Files can be up to 1024K bytes and 32K lines, with up to 512 characters, including new-line, per line (255 for 16-bit machines). *bfs* is usually more efficient than *ed*(1) for scanning a file, since the file is not copied to a buffer. It is most useful for identifying sections of a large file where *csplit*(1) can be used to divide it into more manageable pieces for editing.

Normally, the size of the file being scanned is printed, as is the size of any file written with the **w** command. The optional – suppresses printing of sizes. Input is prompted with * if **P** and a carriage return are typed, as in *ed*(1). Prompting can be turned off again by inputting another **P** and carriage return. Note that messages are given in response to errors if prompting is turned on.

All address expressions described under *ed*(1) are supported. In addition, regular expressions may be surrounded with two symbols besides / and ?: > indicates downward search without wrap-around, and < indicates upward search without wrap-around. There is a slight difference in mark names: only the letters **a** through **z** may be used, and all 26 marks are remembered.

The **e**, **g**, **v**, **k**, **p**, **q**, **w**, =, ! and null commands operate as described under *ed*(1). Commands such as ——, +++–, +++=, –12, and **+4p** are accepted. Note that **1,10p** and **1,10** will both print the first ten lines. The **f** command only prints the name of the file being scanned; there is no *remembered* file name. The **w** command is independent of output diversion, truncation, or crunching (see the **xo, xt** and **xc** commands, below). The following additional commands are available:

xf *file*
: Further commands are taken from the named *file*. When an end-of-file is reached, an interrupt signal is received or an error occurs, reading resumes with the file containing the **xf**. The **xf** commands may be nested to a depth of 10.

xn
: List the marks currently in use (marks are set by the **k** command).

xo [*file*]

Further output from the **p** and null commands is diverted to the named *file*, which, if necessary, is created mode 666 (readable and writable by everyone), unless your *umask* setting (see *umask*(1)) dictates otherwise. If *file* is missing, output is diverted to the standard output. Note that each diversion causes truncation or creation of the file.

: *label*

This positions a *label* in a command file. The *label* is terminated by new-line, and blanks between the **:** and the start of the *label* are ignored. This command may also be used to insert comments into a command file, since labels need not be referenced.

(. , .)xb/*regular expression*/*label*

A jump (either upward or downward) is made to *label* if the command succeeds. It fails under any of the following conditions:

1. Either address is not between **1** and **$**.
2. The second address is less than the first.
3. The regular expression does not match at least one line in the specified range, including the first and last lines.

On success, **.** is set to the line matched and a jump is made to *label*. This command is the only one that does not issue an error message on bad addresses, so it may be used to test whether addresses are bad before other commands are executed. Note that the command

```
xb/^/ label
```

is an unconditional jump.
The **xb** command is allowed only if it is read from someplace other than a terminal. If it is read from a pipe only a downward jump is possible.

xt *number*

Output from the **p** and null commands is truncated to at most *number* characters. The initial number is 255.

xv[*digit*] [*spaces*] [*value*]

The variable name is the specified *digit* following the **xv**. The commands **xv5100** or **xv5 100** both assign the value **100** to the variable **5**. The command **xv61,100p** assigns the value **1,100p** to the variable **6**. To reference a variable, put a **%** in front of the variable name. For example, using the above assignments for variables **5** and **6**:

```
1,%5p
1,%5
%6
```

will all print the first 100 lines.

```
g/%5/p
```

would globally search for the characters **100** and print each line containing a match. To escape the special meaning of **%**, a \ must precede it.

```
g/".*\%[cds]/p
```

could be used to match and list lines containing *printf* of characters, decimal integers, or strings.

Another feature of the **xv** command is that the first line of output from a UNIX system command can be stored into a variable. The only requirement is that the first character of *value* be an **!**. For example:

```
.w junk
xv5!cat junk
!rm junk
!echo "%5"
xv6!expr %6 + 1
```

would put the current line into variable **5**, print it, and increment the variable **6** by one. To escape the special meaning of **!** as the first character of *value*, precede it with a \.

```
xv7\!date
```

stores the value **!date** into variable **7**.

xbz *label*

xbn *label*

These two commands will test the last saved *return code* from the execution of a UNIX system command (**!***command*) or nonzero value, respectively, to the specified label. The two examples below both search for the next five lines containing the string size.

```
xv55
: 1
/size/
xv5!expr %5 – 1
!if 0%5 != 0 exit 2
xbn 1
xv45
: 1
/size/
xv4!expr %4 – 1
!if 0%4 = 0 exit 2
xbz 1
```

xc [*switch*]

If *switch* is **1**, output from the **p** and null commands is crunched; if *switch* is **0** it is not. Without an argument, **xc** reverses *switch*. Initially *switch* is set for no crunching. Crunched output has strings of tabs and blanks reduced to one blank and blank lines suppressed.

SEE ALSO

csplit(1), ed(1), umask(1).

DIAGNOSTICS

? for errors in commands, if prompting is turned off. Self-explanatory error messages when prompting is on.

NAME

bindkey – function key binding facility for use with *wsh*(1G)

SYNOPSIS

bindkey [–r key[,binding] ...]
bindkey [–l key[,binding] ...]

DESCRIPTION

bindkey is a program which provides an interface to the *wsh*(1G) function key binding facilities.

key is the name of a key on the keyboard; type *bindkey* without arguments to obtain a list of valid keys. The following are valid *bindkey* keys:

f1	**f2**	**f3**	**f4**	**f5**	**f6**
f7	**f8**	**f9**	**f10**	**f11**	**f12**
print-scrn		**scroll-lock**		**pause**	
insert	**home**	**page-up**			
end	**page-down**		**left-arrow**		
up-arrow		**down-arrow**		**right-arrow**	

binding is the text string which the key is bound to. The text of the binding argument must be in the printable character set. Using the "\" character, other character codes can be made a part of the binding. It is important to remember that the binding text is interpreted by the shell you are using before it is passed to *bindkey* on the command line. For remote bindings, the text is interpreted a second time when the key bound to the text is pressed. See the *csh*(1) and *sh*(1) man pages for rules concerning escape sequences for special characters, such as newline (\n).

The –r option binds the text string to the given key. When the key is pressed, the data is sent to the process that *wsh* is managing (such as the shell).

The **-l** option performs the same binding as **-r**, except that the text string defines a function internal to *wsh.* When the key is pressed, the local binding is executed by *wsh* directly, and not passed on to the program that *wsh* is managing.

The set of local *wsh* functions and their keywords are listed below.

copy	copy the current selection, if any, to the cut buffer.
down-line	scroll the view one line, as if the down arrow button on the scroll bar was clicked.

down-page	scroll the view down one page, as if the up arrow button on the scroll bar was clicked while the Shift key was pressed.
end	scroll the view to the bottom of the *wsh* contents, as if the scroll bar was clicked at the bottom.
home	scroll the view to the top of the *wsh* contents, as if the scroll bar was clicked at the top.
pop	pop the window to the front.
push	push the window to the back.
send	send the contents of the cut buffer to the process that *wsh* is managing. The data is sent as if the user typed it.
toggle-redirect	Toggle the output redirect (only valid when *wsh* is run with the -R option). If the *wsh* output was redirected to the secondary device, the output is redirected back to the *wsh* window. If the output was directed to the *wsh* window, redirect it to the secondary device.
up-line	scroll the view one line, as if the up arrow button on the scroll bar was clicked.
up-page	scroll the view up one page, as if the up arrow button on the scroll bar was clicked while the Shift key was pressed.

wsh has the following default local bindings (see above):

The **f3** key is bound to the local **copy** function.

The **f4** key is bound to the local **send** function.

The **home** key is bound to the local **home** function.

The **end** key is bound to the local **end** function.

The **page-up** key is bound to the local **up-page** function.

The **page-down** key is bound to the local **down-page** function.

If no *binding* is provided, then *bindkey* will restore the key to its default global binding.

EXAMPLES

The following example of remote binding performs the command 'ls -l' when the F1 key is pressed:

```
bindkey -r f1,'ls -l\n'
```

The following example of local binding scrolls the *wsh* display up one line when the up-arrow on the cursor control pad is pressed.

```
        bindkey -l up-arrow,up-line
```

The following example reverts the binding on the F1 function key to the system default:

```
        bindkey -r f1
```

WARNING

bindkey provides a restricted interface which will be adjusted when the underlying key binding facility is completed.

BUGS

There is currently no way to query a binding. There is no information around that describes what the "original default global bindings" are.

SEE ALSO

wsh (1G), sh (1), csh (1)

NAME

bitmap, bmtoa, atobm – bitmap editor and converter utilities for X

SYNOPSIS

bitmap [-options ...] *filename WIDTHxHEIGHT*

bmtoa [-chars ...] [*filename*]

atobm [-chars *cc*] [-name *variable*] [-xhot *number*] [-yhot *number] [filename]*

DESCRIPTION

The *bitmap* program is a rudimentary tool for creating or editing rectangular images made up of 1's and 0's. Bitmaps are used in X for defining clipping regions, cursor shapes, icon shapes, and tile and stipple patterns.

The *bmtoa* and *atobm* filters convert *bitmap* files (FILE FORMAT) to and from ASCII strings. They are most commonly used to quickly print out bitmaps and to generate versions for including in text.

USAGE

Bitmap displays grid in which each square represents a single bit in the picture being edited. Squares can be set, cleared, or inverted directly with the buttons on the pointer and a menu of higher level operations such as draw line and fill circle is provided to the side of the grid. Actual size versions of the bitmap as it would appear normally and inverted appear below the menu.

If the bitmap is to be used for defining a cursor, one of the squares in the images may be designated as the *hotspot*. This determines where the cursor is actually pointing. For cursors with sharp tips (such as arrows or fingers), this is usually at the end of the tip; for symmetric cursors (such as crosses or bullseyes), this is usually at the center.

Bitmaps are stored as small C code fragments suitable for including in applications. They provide an array of bits as well as symbolic constants giving the width, height, and hotspot (if specified) that may be used in creating cursors, icons, and tiles.

The *WIDTHxHEIGHT* argument gives the size to use when creating a new bitmap (the default is 16x16). Existing bitmaps are always edited at their current size.

If the *bitmap* window is resized by the window manager, the size of the squares in the grid will shrink or enlarge to fit.

OPTIONS

Bitmap accepts the following options:

–help
This option will cause a brief description of the allowable options and parameters to be printed.

–display *display*
This option specifies the name of the X server to used.

–geometry *geometry*
This option specifies the placement and size of the bitmap window on the screen. See *X* for details.

–nodashed
This option indicates that the grid lines in the work area should not be drawn using dashed lines. Although dashed lines are prettier than solid lines, on some servers they are significantly slower.

–name *variablename*
This option specifies the variable name to be used when writing out the bitmap file. The default is to use the basename of the *filename* command line argument.

–bw *number*
This option specifies the border width in pixels of the main window.

–fn *font*
This option specifies the font to be used in the buttons.

–fg *color*
This option specifies the color to be used for the foreground.

–bg *color*
This option specifies the color to be used for the background.

–hl *color*
This option specifies the color to be used for highlighting.

–bd *color*
This option specifies the color to be used for the window border.

–ms *color*
This option specifies the color to be used for the pointer (mouse).

Bmtoa accepts the following option:

–chars *cc*
This option specifies the pair of characters to use in the string version of the bitmap. The first character is used for 0 bits and the second character is used for 1 bits. The default is to use dashes (-) for 0's and sharp signs (#) for 1's.

Atobm accepts the following options:

–chars *cc*
This option specifies the pair of characters to use when converting string bitmaps into arrays of numbers. The first character represents a 0 bit and the second character represents a 1 bit. The default is to use dashes (-) for 0's and sharp signs (#) for 1's.

–name *variable*
This option specifies the variable name to be used when writing out the bitmap file. The default is to use the basename of the *filename* command line argument or leave it blank if the standard input is read.

–xhot *number*
This option specifies the X coordinate of the hotspot. Only positive values are allowed. By default, no hotspot information is included.

–yhot *number*
This option specifies the Y coordinate of the hotspot. Only positive values are allowed. By default, no hotspot information is included.

CHANGING GRID SQUARES

Grid squares may be set, cleared, or inverted by pointing to them and clicking one of the buttons indicated below. Multiple squares can be changed at once by holding the button down and dragging the cursor across them. Set squares are filled and represent 1's in the bitmap; clear squares are empty and represent 0's.

Button 1
This button (usually leftmost on the pointer) is used to set one or more squares. The corresponding bit or bits in the bitmap are turned on (set to 1) and the square or squares are filled.

Button 2
This button (usually in the middle) is used to invert one or more squares. The corresponding bit or bits in the bitmap are flipped (1's become 0's and 0's become 1's).

Button 3
This button (usually on the right) is used to clear one or more squares. The corresponding bit or bits in the bitmap are turned off (set to 0) and the square or squares are emptied.

MENU COMMANDS

To make defining shapes easier, *bitmap* provides 13 commands for drawing whole sections of the grid at once, 2 commands for manipulating the hotspot, and 2 commands for updating the bitmap file and exiting. A command buttons for each of these operations is located to the right of the grid.

Several of the commands operate on rectangular portions of the grid. These areas are selected after the command button is pressed by moving the cursor to the upper left square of the desired area, pressing a pointer button, dragging the cursor to the lower right hand corner (with the button still pressed) , and then releasing the button. The command may be aborted by pressing any other button while dragging or by releasing outside the grid.

To invoke a command, move the pointer over that command and click any button.

Clear All
: This command is used to clear all of the bits in the bitmap as if Button 3 had been dragged through every square in the grid. It cannot be undone.

Set All
: This command is used to set all of the bits in the bitmap as if Button 1 had been dragged through every square in the grid. It cannot be undone.

Invert All
: This command is used to invert all of the bits in the bitmap as if Button 2 had been dragged through every square in the grid.

Clear Area
: This command is used to clear a region of the grid as if Button 3 had been dragged through each of the squares in the region. When this command is invoked, the cursor will change shape to indicate that the area to be cleared should be selected as outlined above.

Set Area
: This command is used to set a region of the grid as if Button 1 had been dragged through each of the squares in the region. When this command is invoked, the cursor will change shape to indicate that the area to be set should be selected as outlined above.

Invert Area
: This command is used to inverted a region of the grid as if Button 2 had been dragged through each of the squares in the region. When this command is invoked, the cursor will change shape to indicate that the area to be inverted should be selected as outlined above.

Copy Area

This command is used to copy a region of the grid from one location to another. When this command is invoked, the cursor will change shape to indicate that the area to be copied should be selected as outlined above. The cursor should then be clicked on the square to which the upper left hand corner of the region should be copied.

Move Area

This command is used to move a region of the grid from one location to another. When this command is invoked, the cursor will change shape to indicate that the area to be moved should be selected as outlined above. The cursor should then be clicked on the square to which the upper left hand corner of the region should be moved. Any squares in the region's old position that aren't also in the new position are cleared.

Overlay Area

This command is used to copy all of the set squares in a region of the grid from one location to another. When this command is invoked, the cursor will change shape to indicate that the area to be copied should be selected as outlined above. The cursor should then be clicked on the square to which the upper left hand corner of the region should be overlaid. Only the squares that are set in the region will be touched in the new location.

Line

This command will set the squares in a line between two points. When this command is invoked, the cursor will change shape to indicate that the pointer should be clicked on the two end points of the line.

Circle

This command will set the squares on a circle specified by a center and a point on the curve. When this command is invoked, the cursor will change shape to indicate that the pointer should be clicked on the center of the circle and then over a point on the curve. Small circles may not look very round because of the size of the grid and the limits of having to work with discrete pixels.

Filled Circle

This command will set all of the squares in a circle specified by a center and a point on the curve. When this command is invoked, the cursor will change shape to indicate that the pointer should be clicked on the center of the circle and then over a point on the curve. All squares side and including the circle are set.

Flood Fill
This command will set all clear squares in an enclosed shape. When this command is invoked, the cursor will change shape to indicate that the pointer should be clicked on any empty square inside the shape to be filled. All empty squares that border horizontally or vertically with the indicated square are set out to the enclosing shape. If the shape is not closed, the entire grid will be filled.

Set Hot Spot
This command designates one square in the grid as the hot spot if this bitmap to be used for defining a cursor. When the command is invoked, the cursor will change indicating that the pointer should be clicked on the square to contain the hot spot.

Clear Hot Spot
This command removes any designated hot spot from the bitmap.

Write Output
This command writes a small fragment of C code representing the bitmap to the filename specified on the command line. If the file already exists, the original file will be renamed to *filename~* before the new file is created. If an error occurs in either the renaming or the writing of the bitmap file, a dialog box will appear asking whether or not *bitmap* should use */tmp/filename* instead.

Quit
This command causes *bitmap* to display a dialog box asking whether or not it should save the bitmap (if it has changed) and then exit. Answering *yes* is the same as invoking *Write Output*; *no* causes *bitmap* to simply exit; and *cancel* will abort the *Quit* command so that more changes may be made.

FILE FORMAT

The *Write Output* command stores bitmaps as simple C program fragments that can be compiled into programs, referred to by X Toolkit pixmap resources, manipulated by other programs (see *xsetroot*), or read in using utility routines in the various programming libraries. The width and height of the bitmap as well as the hotspot, if specified, are written as preprocessor symbols at the start of the file. The bitmap image is then written out as an array of characters:

```
#define name_width 11
#define name_height 5
#define name_x_hot 5
#define name_y_hot 2

static char name_bits[] = {
   0x91, 0x04, 0xca, 0x06, 0x84,
   0x04, 0x8a, 0x04, 0x91, 0x04
};
```

The **name** prefix to the preprocessor symbols and to the bits array is constructed from the *filename* argument given on the command line. Any directories are stripped off the front of the name and any suffix beginning with a period is stripped off the end. Any remaining non-alphabetic characters are replaced with underscores. The *name_x_hot* and *name_y_hot* symbols will only be present if a hotspot has been designated using the *Set Hot Spot* command.

Each character in the the array contains 8 bits from one row of the image (rows are padded out at the end to a multiple of 8 to make this is possible). Rows are written out from left to right and top to bottom. The first character of the array holds the leftmost 8 bits of top line, and the last characters holds the right most 8 bits (including padding) of the bottom line. Within each character, the leftmost bit in the bitmap is the least significant bit in the character.

This process can be demonstrated visually by splitting a row into words containing 8 bits each, reversing the bits each word (since Arabic numbers have the significant digit on the right and images have the least significant bit on the left), and translating each word from binary to hexadecimal.

In the following example, the array of 1's and 0's on the left represents a bitmap containing 5 rows and 11 columns that spells *X11*. To its right is is the same array split into 8 bit words with each row padded with 0's so that it is a multiple of 8 in length (16):

```
10001001001        10001001 00100000
01010011011        01010011 01100000
00100001001        00100001 00100000
01010001001        01010001 00100000
10001001001        10001001 00100000
```

Reversing the bits in each word of the padded, split version of the bitmap yields the left hand figure below. Interpreting each word as hexadecimal number yields the array of numbers on the right:

```
10010001 00000100      0x91 0x04
11001010 00000110      0xca 0x06
10000100 00000100      0x84 0x04
10001010 00000100      0x8a 0x04
10010001 00000100      0x91 0x04
```

The character array can then be generated by reading each row from left to right, top to bottom:

```
static char name_bits[] = {
  0x91, 0x04, 0xca, 0x06, 0x84,
  0x04, 0x8a, 0x04, 0x91, 0x04
};
```

The *bmtoa* program may be used to convert *bitmap* files into arrays of characters for printing or including in text files. The *atobm* program can be used to convert strings back to *bitmap* format.

USING BITMAPS IN PROGRAMS

The format of *bitmap* files is designed to make bitmaps and cursors easy to use within X programs. The following code could be used to create a cursor from bitmaps defined in *this.cursor* and *this_mask.cursor*:

```
#include "this.cursor"
#include "this_mask.cursor"

XColor foreground, background;
/* fill in foreground and background color structures */
Pixmap source = XCreateBitmapFromData (display, drawable,
      this_bits, this_width, this_height);
Pixmap mask = XCreateBitmapFromData (display, drawable,
      this_mask_bits, this_mask_width, this_mask_height);
Cursor cursor = XCreatePixmapCursor (display, source, mask,
      foreground, background, this_x_hot, this_y_hot);
```

Additional routines are available for reading in *bitmap* files and returning the data in the file, in Bitmap (single-plane Pixmap for use with routines that require stipples), or full depth Pixmaps (often used for window backgrounds and borders). Applications writers should be careful to understand the difference between Bitmaps and Pixmaps so that their programs function correctly on color and monochrome displays.

For backward compatibility, *bitmap* will also accept X10 format *bitmap* files. However, when the file is written out again it will be in X11 format

X DEFAULTS

Bitmap uses the following resources:

Background
The window's background color. Bits which are 0 in the bitmap are displayed in this color. This option is useful only on color displays. The default value is *white*.

BorderColor
The border color. This option is useful only on color displays. The default value is *black*.

BorderWidth
The border width. The default value is 2.

BodyFont
The text font. The default value is *variable*.

Dashed
If ''off'', then *bitmap* will draw the grid lines with solid lines. The default is ''on''.

Foreground
The foreground color. Bits which are 1 in the bitmap are displayed in this color. This option is useful only on color displays. The default value is *black*.

Highlight
The highlight color. *bitmap* uses this color to show the hot spot and to indicate rectangular areas that will be affected by the *Move Area, Copy Area, Set Area,* and *Invert Area* commands. If a highlight color is not given, then *bitmap* will highlight by inverting. This option is useful only on color displays.

Mouse
The pointer (mouse) cursor's color. This option is useful only on color displays. The default value is *black*.

Geometry
The size and location of the bitmap window.

Dimensions
The *WIDTHxHEIGHT* to use when creating a new bitmap.

SEE ALSO

X(1), *Xlib - C Language X Interface* (particularly the section on *Manipulating Bitmaps*), *XmuReadBitmapDataFromFile*

BUGS

The old command line arguments aren't consistent with other X programs.

If you move the pointer too fast while holding a pointer button down, some squares may be missed. This is caused by limitations in how frequently the X server can sample the pointer location.

There is no way to write to a file other than the one specified on the command line.

There is no way to change the size of the bitmap once the program has started.

There is no *undo* command.

COPYRIGHT

AUTHOR

bitmap by Ron Newman, MIT Project Athena; documentation, *bmtoa*, and *atobm* by Jim Fulton, MIT X Consortium.

NAME

blanktime – set the screen blanking timeout

SYNOPSIS

blanktime nframes

DESCRIPTION

The screen of the iris will normally blank (turn black) if the system receives no input events for about 15 minutes. This is done to protect the color display. *blanktime* makes it possible to change the time before the screen is blanked, or disable the screen blanking feature completely. The argument *nframes* specifies the screen blanking timeout in frame times. Since there are 67 frames per second on the standard 60 Hz monitor, *nframes* should be 67 times the number of seconds that the system should wait before blanking the screen. If *nframes* is given as 0, screen blanking is disabled.

EXAMPLE

```
blanktime 6700
```

makes the screen blank if no input device is touched for 100 seconds.

NAME

bldfamily – build font family description

SYNOPSIS

bldfamily [–**d***dirname*] [–**o***outname*] [–**v**] *names*

DESCRIPTION

bldfamily scans a sets of NeWS font files and produces a NeWS font family file. A font family is a set of font files that are grouped together to provide a single PostScript font. In PostScript, a font has a name, like Times-Roman, and can be rendered in many different sizes. A NeWS font file is an instance of a PostScript font at a particular size. Font family files contain the information necessary for NeWS to pick the right bitmap font.

bldfamily scans directory *dirname* for files with extension ".font" or ".metrics" and where the leading non-digit characters of the filename match one of the *names* . The family file that is built will be written to *dirname*/*outname*.**ffam.**

If *outname* isn't specified, it defaults to the first of the *names* . If *dirname* isn't specified, it defaults to "$FONDIR" if defined, "." otherwise.

OPTIONS

–**d***dirname*	Specifies the directory to scan and put the **.ffam** file into.
–**o***outname*	Specifies the output font name
–**v**	Verbose – gives a more detailed description of what is going on.

EXAMPLE

```
% dumpfont -d /usr/newfonts -n Boston boston*.vfont
% bldfamily -d/usr/newfonts Boston
```

The first command calls *dumpfont* and converts all the vfont files whose names match "boston*.vfont" into NeWS format. It puts them into */usr/newfonts* and changes their font name to Boston (it would have defaulted to boston). The second command calls *bldfamily* and scans */usr/newfonts* for "Boston*.font" and builds a font family file for them, which will be called */usr/newfonts/Boston.ffam.*

SEE ALSO

dumpfont(1)

BUGS

The options and directory defaulting should be consistent with *dumpfont* . *bldfont* is clumsy to invoke.

NAME

bru – backup and restore utility

SYNOPSIS

bru modes [control options] [selection options] files

DESCRIPTION

Bru is a **Unix** filesystem backup utility with significant enhancements over other more common utilities such as *tar*, *cpio*, *volcopy*, and *dd*. Some of *bru*'s capabilities include:

- Full or incremental backup with quick and easy restoration of files.
- Multiple physical volumes per archive.
- Data integrity assurance via checksum computation on every archive block.
- Ability to properly save and restore directories, symbolic links, block special files, and character special files.
- Comparison of archives with current directory hierarchy.
- Ability to recover files from corrupted archives or damaged media with minimal data loss.
- No inherent maximum archive buffer size.
- Improved performance through random access archive I/O when available.
- Automatic byte or half word swapping as necessary when reading archives produced on other machines.
- Recognition of filename generation patterns in the same form as the shell for files read from an archive.

When *files* are specified on the command line then the actions to be performed are limited to those *files*. If a named file is a directory then it and all its descendents are used. If no *files* are specified then the default for writing archives is all files in and below the current directory. The default for reading archives is selection of all files in the archive.

If ''–'' is given instead of *files* then the standard input is read to obtain the file list. This is useful in conjunction with the *find* command to provide finer control over files selected for backup. Obviously this mode is only valid when *bru* is not also reading its archive from the standard input.

DEFAULTS

Various default parameters, such as archive device name and size, archive buffer size, controlling terminal name, etc. are system dependent. These defaults, along with version, variant, and other miscellaneous internal information may be discovered via the **–h** mode.

bru assumes the default tape device to be **/dev/tape** if no tape device is specified using the **–f** option. For this to work correctly, **/dev/tape** should be linked to the actual tape device, for ex. **/dev/mt/tps0d6**. Also there should be an appropriate entry for **/dev/mt/tps0d6** in the **/etc/brutab** file.

MODES

One or more of the following modes must be specified. The order of execution, from highest priority to lowest, is **ecitxdgh**.

–c **Create** a new archive. Forces a new archive to be created regardless of whether one currently exists. Writing starts at the first block.

–d **Differences** between archived *files* and current *files* are detected and reported. May be specified more than once, as **–dd –ddd** or **–dddd** to control level of difference checking.

When specified as **–d** *bru* reports when it discovers that a regular file's size (st_size) or contents (when compared as byte streams) has changed since the archive was made.

When specified as **–dd** *bru* reports additional differences in modification date (st_mtime) access mode (st_mode), number of links (st_nlink) for non-directory files, differences in the contents of symbolic links, owner id (st_uid), and group id (st_gid).

When specified as **–ddd** *bru* reports additional differences in host device (st_dev), major/minor device (st_rdev) for special files, and time of last access (st_atime) for regular files.

When specified as **–dddd** *bru* reports all differences except time of last status change (st_ctime is not resettable), major/minor device numbers for non-special files (meaningless), and size differences for directory files (may have empty entries). The **–dddd** mode is generally only meaningful during a verification pass with full backups of quiescent filesystems.

–e **Estimate** media requirements for archive creation with same arguments. Prints estimated number of volumes, number of files to be archived, total number of archive blocks, and total size of archive in kilobytes. If the media size is unknown or unspecified, it is assumed to be infinite.

−g Dump archive info block in a form more easily parsed by programs implementing a complete filesystem management package. Performs no other archive actions.

−h Print **help** summary of options. Also prints some internal information such as version number and default values for archive pathname, media size, archive buffer size, etc.

−i **Inspect** archive for internal consistency and data integrity. When −vv option is also given, prints information from archive header block.

−t List **table** of contents of archive. When used with the −v option will give a verbose table of contents in the same format as the ''ls −l'' command. When used with the −vv option will also indicate what files are linked to other files, and where symbolic links point to.

−x **Extract** named *files* from archive. If an archived file is extracted (see **−u** option) then the access mode, device id (special files only), owner uid, group uid, access time, and modification time are also restored. If the −C flag is given (see below), then the owner uid and group uid will be changed to that of the current user.

Nonexistent directories are recreated from archived directories if possible, otherwise they are created with appropriate defaults for the current user. Extracted or created directories are initially empty.

CONTROL OPTIONS

Many of the control options are similar in function to their *tar* or *cpio* equivalents.

Sizes are specified in bytes. The scale factors M, k, or b, can be used to indicate multiplication by 2**20, 1024, or 512 respectively. Thus ''10k'', ''20b'', and ''10240'' all specify the same number of bytes.

−# str Use string *str* as a control string for the built in debugging system. This option provides information about the internal workings of *bru* for the software maintainer or the merely curious. Some examples are given later.

−a Do not update the access times of disk files that have been read while performing other actions. Normally *bru* restores the access and modification times of disk files after they have been read to the access and modification times to whatever they were before the read operation. Restoring the times prevents defeat of the mechanism used to track down and remove ''dead'' files that haven't been accessed in any meaningful way recently.

−b *bsize* Use *bsize* as the archive input/output **buffer** size. The minimum is the size of an archive block (2k or 2048 bytes) and the maximum is determined by available memory and I/O device limitations. If *bsize* is not an even multiple of 2048 bytes, it will be rounded up. Normally this option is only required with the **−c** mode since *bru* writes this information in the archive header block. If specified, *bsize* overrides any existing default value (generally 20k), whether built in or read from the archive header.

−B Useful in shell scripts where *bru* is run in the **background** with no operator present. Under these conditions, *bru* simply terminates with appropriate error messages and status, rather than attempting interaction with the terminal.

−C Change the owner (**chown**) and group of each extracted file to the owner uid and group gid of the current user. Normally, *bru* will restore the owner and group to those recorded in the archive. This flag causes bru to follow the system default, with extracted files having the same owner and group as the user running *bru* , including Root.

The −C option is useful with archives imported from other systems. In general, it should not be used by the operator or system administrator when restoring saved files. Use the **−tv** option to see the owner and group of files stored in the archive.

−D Causes *bru* to use double buffering to the archive device on systems that have System V style shared memory. Depending upon hardware constraints, double buffering may dramatically increase the archive device I/O rate, but may adversely affect the error recovery algorithms.

–f *path* Use *path* as the archive **file** instead of the default. If the *path* is "–" then *bru* uses the standard input for archive reading or standard output for archive writing, as appropriate. If the path is remote then it needs to be specified as *[user@]system:/dev/???* The <user@> part of the path is optional, and if none is specified then the user's login is used. The login has to be equivalently mapped to the remote machine if it has a password.

–F **Fast** mode. In fast mode, checksum computations and comparisons are disabled. This mode is useful when the output of one *bru* is piped to the input of another *bru*, or when the data integrity of the archive transmission medium is essentially perfect. Archives recorded with fast mode enabled must also be read with fast mode. Also, be aware that some of the automatic features of *bru*, such as automatic byte swapping, are not functional in fast mode.

–j Convert **absolute** pathnames to "." relative pathnames. This option applies only to the –x mode.

–X Echo relative pathnames as absolute pathnames. This option only valid when used in conjunction with the –x, –j and –v options.

–L *str* **Label** the archive with the specified string *str*. *Str* is limited to 63 characters and is usually some meaningful reminder pertaining to the archive contents.

–l Ignore unresolved **links**. Normally bru reports problems with unresolved links (both regular and symbolic links). This option suppresses all such complaints.

–m Do not cross **mounted** file system boundaries during expansion of explicitly named directories. This option applies only to directories named in *files*. It limits selection of directory descendents to those located on the same filesystem as the explicitly named directory. This option currently applies only to the –c and –e modes.

–p **Pass** over files in archive by reading rather than seeking. Normally *bru* will use random access capabilities if available. This option forces reading instead of seeks.

−R **Remote** files are to be **excluded** from the archive. If the system does not support remote filesystems, this option is ignored.

−s *msize* Use *msize* as the media **size**. The effective media **size** will be computed from *msize* since it must be integral multiple of the input/output buffer size (see the **−b** option). Normally this option is only required with the **−c** mode since *bru* writes this information in the archive header block. If specified, *msize* overrides any existing default value, whether built in or read from the archive header.

−v Enable **verbose** mode. May be specified more than once, as **−vv**, **−vvv**, or **−vvvv**, to get even more verbosity.

−w **Wait** for confirmation. *Bru* will print the file name, the action to be taken, and **wait** for confirmation. Any response beginning with 'y' will cause the action to complete. Any other response will abort the action.

−Z Use 12-bit LZW file compression. This is not the default because not all versions of *bru* know how to deal with compressed files. When the **−v** option is also selected, the compression ratio for each file is printed as a percentage. When this flag is used in conjunction with the **−t** option on an archive that contains compressed file, the actual archive file sizes and names are printed, rather than the original values before archiving.

FILE SELECTION OPTIONS

The file selection options control which files are selected for processing. Note that some options are only valid with specific modes.

−n *date* Select only files **newer** than *date*. The *date* is given in one of the forms:

DD−MMM−YY[,HH:MM:SS]	EX: 12−Mar−84,12:45:00
MM/DD/YY[,HH:MM:SS]	EX: 3/12/84
MMDDHHMM[YY]	EX: 0312124584
pathname	EX: /etc/lastfullbackup

The time of day is optional in the first two forms. If present, it is separated from the date with a comma.

If *date* is really the pathname of a file, then the modification date of that file will be used instead. This is useful in automated backups when a dummy file is ''touched'' to save the date of last backup.

−o *user* Select only files **owned** by *user*. *User* may be specified in one of three ways:

- As an ascii string corresponding to a user name in the password file.
- As the pathname of a file in which case the owner of that file is used.
- As a numeric value (decimal).

−u *flags* When used in conjunction with −x mode, causes files of type specified by *flags* to be **unconditionally** selected regardless of modification times. Normally *bru* will not overwrite (supersede) an existing file with an older archive file of the same name. Files which are not superseded will give warnings if **verbose** mode level 2 (−vv) or higher is enabled. Possible characters for *flags* are:

b	select block special files
c	select character special files
d	select directories
l	select symbolic links
p	select fifos (named pipes)
r	select regular files

Selection of directories only implies that their attributes may be modified. Existing directories are never overwritten, this option merely allows their attributes to be set back to some previously existing state.

Selection of symbolic links only implies that the contents of the link will be modified. It is currently impossible to change access time, modification time, or the file mode of a symbolic link.

EXAMPLES

Create (−c) a new archive of all files under ''/usr/src'', writing archive to file (−f) ''/dev/rmt0'' using multiple tapes with a maximum size (−s) of 30 megabytes per tape.

```
bru -c -f /dev/rmt0 -s 30M /usr/src
```

Create (**–c**) a new archive on the default device in the first pass, archiving all files in and below the current directory which have been created or modified (**–n**) since 3 P.M. on 14–Jan–84. Then do a second pass to verify that there are no differences (**–d**) between the archive and current files. Each file is listed (**–v**) as it is processed.

```
bru -cvd -n 14-Jan-84,15:00:00
```

Archive all files owned (**–o**) by user ''user1'' using the default archive device.

```
find / -user user1 -print | bru -c -
bru -c -o user1 /
```

Copy a directory hierarchy from ''/usr/u1'' to ''/usr/u2''.

```
(cd /usr/u1; bru -cf - ) | (cd /usr/u2; bru -xf -)
```

Extract (**–x**) the regular file ''/usr/guest/myfile'' unconditionally (**–ur**) from an archive on file (**–f**) ''/dev/rf0''. Since the device size was recorded in the header block, it need not be specified. Note that option arguments do not need to be separated from their corresponding option flag by whitespace.

```
bru -x -ur -f/dev/rf0 ./usr/guest/myfile
```

Extract (**–x**) all C source files in ''/usr/src/cmd'' that have names beginning with characters 'a' through 'm'. Wait (**–w**) for confirmation before extracting each file.

```
bru -xw '/usr/src/cmd/[a-m]*.c'
```

Inspect (**–i**) a previously created archive on the default device, dumping the contents of the header block for inspection (**–vvv**) and verifying internal consistency and data integrity of the archive.

```
bru -ivvv
```

Perform the same function as the last example except enable various features of the built in debugger (when linked in). The debug control string is a string of the form ''–#<opt1>:<opt2>:...'', where each option is either a single flag character or a flag character followed by a comma separated list. Available flag characters are: 'd' enable debugging for list of keywords, 'f' limit debugging to list of function names, 'F' print source file name, 'L' print source file line numbers, 'n' print nesting depth, 'o' redirect output to listed file, 'p' print process name, 't' enable tracing.

```
bru –ivvv –#t
bru –ivvv –#d:t
bru –ivvv –#d,ar_io,verify:F:L
bru –ivvv –#d:f,ar_seek
bru –ivvv –#d:o,trace.out:t:p
```

Back up the entire root filesystem without crossing mounted (**–m**) filesystem boundaries. The archive will be written to file (**–f**) "/dev/rmt0" using an I/O buffer size (**–b**) of 10k bytes. A record of all files processed will be written to file "brulogfile" for future reference.

```
cd /
bru –cvm –f /dev/rmt0 –b 10k >brulogfile
```

DIAGNOSTICS

Most diagnostics are reasonably informative. The most common have to do with meaningless combinations of options, incompatible options, hitting memory or device limits, unresolved file links, trying to archive or restore something to which access is normally denied, or problems with media errors and/or archive corruption.

DEVICE TABLE

Bru contains an internal table of known devices and their characteristics. This table is dynamically loaded from a data file specified by the environment variable **BRUTAB**, or from **/etc/brutab**, or from an internal default description if neither of the preceding is found.

The format of the *bru* data file for loadable devices is as follows. Note that the table MUST contain at least one entry and the first entry is the default archive device.

Also, the table should contain an entry for the standard input and output, with a name of "-".

Entries consist of a device name field, followed by one or more capabilities fields. Entries may span more than one line by escaping the newline at the end of the line with a '\' character ('\' is last character on line before newline). All whitespace (tabs, blanks, newlines, formfeeds) between fields is ignored.

The device name field must be the first field in the entry and has the following form:

```
<device name>|<device name>| ... |<device name>

ex: /dev/rmt0|/dev/rmt1|/dev/rmt2
```

where each device has the same capabilities as the other devices specified (a class of devices).

Each capability field is of the form:

```
<capability name>=<value>  or  <boolean flag>

ex: size=640K  REOPEN  pwerr=EIO
```

Note that there can be no whitespace between the capability name and the value. Numeric values may be given in absolute form or with a trailing scale factor of:

b or B	Blocks (512 bytes)
k or K	Kilobytes (1024 bytes)
m or M	Megabytes (1024 * 1024 bytes)

Error numbers may be given in absolute or symbolic form, as defined in <errno.h>.

Currently used capabilities include:

Name	Type	Meaning
size	numeric	media size in bytes if known, zero otherwise
seek	numeric	minimum seek resolution, zero if no seeks allowed
prerr	numeric	errno for partial reads
pwerr	numeric	errno for partial writes
zrerr	numeric	errno for zero reads
zwerr	numeric	errno for zero writes
frerr	numeric	errno for read of unformatted media
fwerr	numeric	errno for write of unformatted media
wperr	numeric	errno for write protected media

reopen	boolean	close and reopen archive upon media switch
noreopen	boolean	no close and reopen archive upon media switch
tape	boolean	archive device is a tape drive
rawtape	boolean	archive device is a "raw" tape drive
norewind	boolean	closing does not automatically rewind
advance	boolean	read/writes advance media even when errors occur

For instance, the device table for a 4D that is compiled into *bru* is:

```
/dev/mt/ts0d6 \
   size=44032k seek=0 \
   prerr=EIO pwerr=EIO zrerr=ENOSPC zwerr=ENOSPC \
   frerr=ENOSPC fwerr=0 wperr=EROFS \
   reopen tape advance
/dev/mt/ts0d6nr \
   size=44032k seek=0 \
   prerr=EIO pwerr=EIO zrerr=ENOSPC zwerr=ENOSPC \
   frerr=ENOSPC fwerr=0 wperr=EROFS \
   reopen tape norewind advance
- \
   size=0 seek=0 \
   prerr=0 pwerr=0 zrerr=0 zwerr=0 frerr=0 fwerr=0 wperr=0
```

The Personal Iris version contains the following table:

```
/dev/mt/tps0d6 \
   size=0K seek=0 \
   prerr=EIO pwerr=EIO zrerr=ENOSPC zwerr=ENOSPC \
   frerr=ENOSPC fwerr=0 wperr=EROFS \
   reopen tape advance
/dev/mt/tps0d6nr \
   size=0K seek=0 \
   prerr=EIO pwerr=EIO zrerr=ENOSPC zwerr=ENOSPC \
   frerr=ENOSPC fwerr=0 wperr=EROFS \
   norewind reopen tape advance
- \
   size=0 seek=0 \
   prerr=0 pwerr=0 zrerr=0 zwerr=0 frerr=0 fwerr=0 wperr=0
```

SIGNAL HANDLING

Bru normally catches both interrupt (SIGINT) and quit (SIGQUIT). When interrupt is caught during archive creation or extraction, *bru* completes its work on the current file before cleaning up and exiting. This is the normal way of aborting *bru*. When a quit signal is caught an immediate exit is taken.

Note that during file extraction, a quit signal may leave the last file only partially extracted. Similarly, a quit signal during archive writing may leave the archive truncated. When either interrupt or quit is caught at any other time an immediate exit is taken.

ERROR RECOVERY

When properly configured for a given software/hardware environment, bru can recover from most common errors. For example, attempts to use unformatted media are detected, allowing substitution of formatted media. Random blocks in an archive can be deliberately overwritten (corrupted) without affecting *bru*'s ability to recover data from the rest of the archive. When I/O errors are detected, retries are performed automatically. Out of order sequencing on multi-volume archive reads is detected, allowing replacement with the correct volume.

DIRECTORIES

When creating non-incremental archives *bru* automatically archives all directories necessary to fully restore any file from the archive. During extraction, any required directories which do not already exist are restored from the archive if possible, otherwise they are created with appropriate defaults for the current user.

The net result is that restoration from incremental archives (which may not contain all necessary directories), or incremental restoration from full archives (which may skip directories needed later), may result in creation of directories with the default attributes.

WILDCARDS

When **reading** archives *bru* recognizes file name generation patterns in the same format as the shell. This allows greater flexibility in specifying files to be extracted, compared, or listed. As a special extension to shell type expansion, the sense of the match is reversed for patterns that begin with '!'.

Note that the patterns may have to be quoted to prevent expansion by the shell. Also note that patterns are processed independently, without regard for any other patterns that may or may not be present. In particular, "/bin/a* /bin/b*" is equivalent to "/bin/[ab]*", but "/bin/!a* /bin/!b*" is equivalent to "/bin/*", **not** "/bin/![ab]*".

BYTE/WORD SWAPPING

While reading archives produced on other machines, *bru* automatically attempts to perform byte and/or word swapping as necessary.

If no device table is specified, *bru* automatically uses the no-swap tape device, which provides higher performance and compatibility with non-byte swapped tapes from other systems. The Iris 3000 series does not support non-byte swapped tapes, but the automatic byte-swapping capabilities of *bru* will deal with this problem.

REMOTE TAPE DRIVES

Bru allows the use of remote tape drives for the archive device (via the −f option). A remote tape drive file name has the form

[user@]system:/dev/???

where *system* is the remote system, the optional *user* is the login name to use on the remote system if different from the current user's login name, and /dev/??? is the tape drive to use (1600 BPI or 800 BPI, raw or blocked, rewinding or non-rewinding, etc.). In all cases, the user must have the appropriate permissions on the remote system. (See also the CAVEATS section, below.)

EXIT CODES

Bru always returns meaningful status as follows:

0 Normal exit, no errors or warnings.
1 Warnings (or interrupted).
2 Errors (or quit signal).

SEE ALSO

tar(1), cpio(1), rmt(1M).

UNIX SYSTEM INCOMPATIBILITIES

Bru recognizes special file types that may be allowed on one type of Unix system, but not on another. For instance, on a 4.2 BSD system, *bru* will extract fifos as plain files (mode 0666, as modified by the *umask*), and issue an appropriate error message. Usually, *bru* will issue two messages. The first message will be the more descriptive of the two.

Currently, the only different Unix systems that *bru* fully understands are System V, 4.2 BSD, and Pyramid's OSx.

CAVEATS

Pathnames are limited to 127 characters in length. This could become a chronic problem.

Implementation differences complicate the algorithms for automatic detection of end of file on devices. The algorithms can be fooled, hence the –s option.

Special files moved to a machine other than their original host will generally be useless and possibly even dangerous. This goes double for symbolic links.

When extracting files from archives, patterns used to match directories may result in some unnecessary directories being extracted. For example, if the pattern is ''a/*/c'', and the directory ''a/b'' is encountered in the archive, the directory file ''a/b'' will be extracted since it will be needed when (and if) the file ''a/b/c'' is encountered. When in doubt, use the –w option.

In order to be able to efficiently archive needed directories, *bru* builds an image of the directory tree for *files* using dynamically allocated memory. Since there may be at most 5120 characters passed on the command line, it is very unlikely that *bru* will run out of memory while building the tree from command line arguments. This is not true of file lists read from the standard input, particularly on machines with limited address space.

Information about file linkages is also kept in memory. Some linkages may be lost if memory is exhausted.

Since *bru* is owned by root and runs with ''set user id'' to allow it to create directories and special files, it makes every attempt to prevent normal users from archiving or extracting files they would normally not have access to. There may be loopholes. Also note that anyone with physical or electronic access to an archive, and knowledge of the archive structure, can recover any of its contents by writing their own file extraction program.

Directories which have filesystems mounted on them will not be properly archived until the filesystem is unmounted. This is not generally a problem.

Explicitly naming both a directory and one of its descendents will cause the descendent to be archived twice, unless they are on separate filesystems and the **−m** flag is used.

Explicitly naming a file more than once is ineffective.

When reading from the raw magnetic tape file (rmtxxx) *bru* automatically attempts to adjust the I/O buffer size to match that used to record the archive. Under certain circumstances it may fail and require help via the **−b** option.

Using remote tape drives can be slow.

NAME

bstream – many buffered filter

SYNOPSIS

bstream [**-n** bufs] [**-b** size] [**-l** tr] [**-i** file] [**-o** file]

DESCRIPTION

This command is a filter which buffers input from the input file and writes it to the output file. The command is especially useful for reading from or writing to streaming tape drives, where bstream can be used as a speed matching program to keep the tape streaming as much as possible.

If input or output files are not specified, then *bstream* simply copies the standard input to the standard output through multiple buffers. A tape can be written to or read from in this manner, but *bstream* will not be able to handle multiple tape copies. If multiple tapes are involved, explicitly specifying the tape device via the **–i** or **–o** options will enable *bstream* to recover from end of medium indications and request a new tape.

OPTIONS

The following options are supported:

–b *size* This option specifies the size of buffer to use. The default size is 64K bytes, however any size up to the maximum allowable shared memory segment size for the system is allowed.

–n *cnt* This option sets the number of buffers to use. A minimum of two is enforced, while the maximum is limited by the number of shared memory segments which a process may be attached to simultaneously. More than two buffers can improve performance and make the tape more likely to stream. The default value is set to four.

–l This option directs *bstream* to attempt to lock the buffers into physical memory. This will not be possible unless the command is run by the super-user.

–t This option directs *bstream* to attempt to lock its program image and stack in physical memory. This will not be possible unless the command is run by the super user. A by-product of this option is that the nice value of the *bstream* process is decreased, increasing its scheduling priority.

–i *file* This option specifies the input file as an explicit file name, allowing recovery from end-of-medium conditions on tapes. The user will be prompted for a new tape.

–o *file* This option specifies the output file as an explicit file name, allowing recovery from end-of-medium conditions on tapes. The user will be prompted for a new tape.

–r This options reports a summary of the byte throughput achieved by *bstream* for the entire transfer as well as for each individual tape if multiple tapes are involved.

EXAMPLES

To copy a filesystem onto multiple tapes:

```
tar cf - * | bstream -o/dev/tape
```

Recovering an image from tape is just as easy:

```
bstream -i/dev/tape | tar xf -
```

There is a subtle issue involving tapes that should be understood when using *bstream* with a tape device. Usually, a tape device requires that the tape be written in even multiples of the block size. When *bstream* is used as a streaming filter (see the above examples) this is not a problem, since both *tar(1)* and *cpio(1)* block the output on natural tape boundaries. To protect the user, *bstream* will round the final write to tape up to the nearest tape blocksize boundary, fill the residual of the last block with zeros, and print a warning message if rounding up was needed. *bstream* should always be used with a formatter program such as *tar* or *cpio* when writing to a tape device, since rounding up and filling may have unpredictable effects on programs expecting a simple byte stream. No rounding or filling is done when output is not directed to a tape device.

NOTES

bstream runs only on Silicon Graphics 4D series workstations. On an unloaded 4D60, *bstream* can stream the tape 70 to 80% of the time, achieving byte transfer rates around 70K bytes per second.

C – D

NAME

cal – print calendar

SYNOPSIS

cal [[month] year]

DESCRIPTION

cal prints a calendar for the specified year. If a month is also specified, a calendar just for that month is printed. If neither is specified, a calendar for the present month is printed. *Year* can be between 1 and 9999. The *month* is a number between 1 and 12. The calendar produced is that for England and the United States.

EXAMPLES

An unusual calendar is printed for September 1752. That is the month 11 days were skipped to make up for lack of leap year adjustments. To see this calendar, type: **cal 9 1752**

BUGS

The year is always considered to start in January even though this is historically naive.
Beware that "cal 83" refers to the early Christian era, not the 20th century.

NAME

calendar – reminder service

SYNOPSIS

calendar [–]

DESCRIPTION

calendar consults the file **calendar** in the current directory and prints out lines that contain today's or tomorrow's date anywhere in the line. Most reasonable month-day dates such as ''Aug. 24,'' ''august 24,'' ''8/24,'' etc., are recognized, but not ''24 August'' or ''24/8''. On weekends ''tomorrow'' extends through Monday.

When an argument is present, *calendar* does its job for every user who has a file **calendar** in his or her login directory and sends them any positive results by *mail*(1). Normally this is done daily by facilities in the UNIX operating system.

FILES

/usr/lib/calprog	to figure out today's and tomorrow's dates
/etc/passwd	
/tmp/cal*	

SEE ALSO

mail(1).

BUGS

Your calendar must be public information for you to get reminder service.
calendar's extended idea of ''tomorrow'' does not account for holidays.

NAME

cat – concatenate and print files

SYNOPSIS

cat [–**u**] [–**s**] [–**v** [–**t**] [–**e**]] file ...

DESCRIPTION

cat reads each *file* in sequence and writes it on the standard output. Thus:

cat file

prints **file** on your terminal, and:

cat file1 file2 >file3

concatenates **file1** and **file2**, and writes the results in **file3**.

If no input file is given, or if the argument – is encountered, *cat* reads from the standard input file.

The following options apply to *cat*:

–**u**	The output is not buffered. (The default is buffered output.)
–**s**	*cat* is silent about non-existent files.
–**v**	Causes non-printing characters (with the exception of tabs, new-lines and form-feeds) to be printed visibly. ASCII control characters (octal 000 - 037) are printed as ^*n*, where *n* is the corresponding ASCII character in the range octal 100 - 137 (@, A, B, C, . . ., X, Y, Z, [, \,], ^, and _); the DEL character (octal 0177) is printed ^?. Other non-printable characters are printed as **M**-*x*, where *x* is the ASCII character specified by the low-order seven bits.

When used with the –v option, the following options may be used:

–**t**	Causes tabs to be printed as ^I's.
–**e**	Causes a $ character to be printed at the end of each line (prior to the new-line).

The –**t** and –**e** options are ignored if the –**v** option is not specified.

WARNING

Redirecting the output of **cat** onto one of the files being read will cause the loss of the data originally in the file being read. For example, typing:

cat file1 file2 >file1

will cause the original data in **file1** to be lost.

SEE ALSO

cp(1), pg(1), pr(1).

NAME

cb – C program beautifier

SYNOPSIS

cb [–s] [–j] [–l leng] [–t shiftwidth] [file ...]

DESCRIPTION

The *cb* command reads C programs either from its arguments or from the standard input, and writes them on the standard output with spacing and indentation that display the structure of the code. Under default options, *cb* preserves all user new-lines.

cb accepts the following options.

–s	Canonicalizes the code to the style of Kernighan and Ritchie in *The C Programming Language*.
–j	Causes split lines to be put back together.
–l *leng*	Causes *cb* to split lines that are longer than *leng*.
–t *shiftwidth*	tells *cb* to use a soft tab stop which is different from the hard tabs. Tab characters will be used to save space where possible.

SEE ALSO

cc(1).
Kernighan, B. W., and Ritchie, D. M., *The C Programming Language*. Prentice-Hall, 1978.

BUGS

Punctuation that is hidden in preprocessor statements will cause indentation errors.

NAME

cc – MIPS C compiler

SYNOPSIS

cc [option] ... file ...

DESCRIPTION

cc, the MIPS *ucode* C compiler, produces files in the following formats: MIPS object code in MIPS extended *coff* format (the normal result), binary or symbolic *ucode*, *ucode* object files and binary or symbolic assembly language. *cc* accepts several types of arguments:

Arguments whose names end with '.c' are assumed to be C source programs. They are compiled, and each object program is left in the file whose name consists of the last component of the source with '.o' substituted for '.c'. The '.o' file is only deleted when a single source program is compiled and loaded all at once.

Arguments whose names end with '.s' are assumed to be symbolic assembly language source programs. They are assembled, producing a '.o' file.

Arguments whose names end with '.i' are assumed to be C source after being processed by the C preprocessor. They are compiled without being processed by the C preprocessor.

If only *ucode* object files are to be produced (with the –j flag) each C source file is compiled into a *ucode* object file. The *ucode* object file is left in a the file whose name consists of the last component of the source with '.u' substituted for '.c'.

If the highest level of optimization is specified (with the –O3 flag) or only *ucode* object files are to be produced (with the –j flag) each C source file is compiled into a *ucode* object file. The *ucode* object file is left in a file whose name consists of the last component of the source with '.u' substituted for '.c'.

The suffixes described below primarily aid compiler development and are not generally used. Arguments whose names end with '.B', '.O', '.S', and '.M' are assumed to be binary *ucode*, produced by the front-end, optimizer, *ucode* object file splitter and *ucode* merger respectively. Arguments whose names end with '.U' are assumed to be symbolic *ucode*. Arguments whose names end with '.G' are assumed to be binary assembly language, which is produced by the code generator and the symbolic to binary assembler.

Files that are assumed to be binary *ucode*, symbolic *ucode*, or binary assembly language by the suffix conventions are also assumed to have their corresponding symbol table in a file with a '.T' suffix.

cc always defines the C preprocessor symbols **sgi, mips, host_mips, unix, SVR3, SYSTYPE_SYSV, _SYSTYPE_SYSV, _MIPSEB** and **MIPSEB** to the C macro preprocessor and defines the C preprocessor symbols **LANGUAGE_C** and **_LANGUAGE_C** when a '.c' file is being compiled. *cc* will define the C preprocessor symbols **LANGUAGE_ASSEMBLY** and **_LANGUAGE_ASSEMBLY** when an '.s' file is being compiled. The non-leading-underbar forms are being supplemented in some cases with leading-underbar forms. The old forms violate ANSI C rules. One can see (and check) these definitions with the −v option to *cc* . The C preprocessor adds other standard definitions of its own (see *cpp*(1)).

The following options are interpreted by *cc*. See *ld*(1) for load-time options.

−c Suppress the loading phase of the compilation and force an object file to be produced even if only one program is compiled.

−g0 Have the compiler produce no symbol table information for symbolic debugging. This is the default.

−g1 Have the compiler produce additional symbol table information for accurate but limited symbolic debugging of partially optimized code. This option overrides the optimization options (**−O, −O1, −O2, −O3**).

−g or **−g2**
Have the compiler produce additional symbol table information for full symbolic debugging and not do optimizations that limit full symbolic debugging. These options override the optimization options (**−O, −O1, −O2, −O3**).

−g3 Have the compiler produce additional symbol table information for full symbolic debugging for fully optimized code. This option makes the debugger inaccurate. This option can be used with the optimization options (**−O, −O1, −O2, −O3**).

−w Suppress warning messages.

−p0 Do not permit any profiling. This is the default. If loading happens, the standard runtime startup routine (*crt1.o*) is used; no profiling library is searched.

−p or **−p1**
Set up for profiling by periodically sampling the value of the program counter. This option only effects the loading. When loading happens, this option replaces the standard runtime startup routine with the profiling runtime startup routine (*mcrt1.o*) and searches the level 1 profiling library (*libprof1.a*) When profiling happens, the startup routine calls *monstartup*(3) and produces a

file *mon.out* that contains execution-profiling data for use with the postprocessor *prof*(1). NOTE: **−p** and **−p1** may not be used with programs linked to shared libraries.

−O0 Turn off all optimizations.

−O1 Turn on all optimizations that can be done quickly. This is the default.

−O or **−O2**
Invoke the global *ucode* optimizer.

−O3 Do all optimizations, including global register allocation. This option must precede all source file arguments. With this option, a *ucode* object file is created for each C source file and left in a '.u' file. The newly created *ucode* object files, the *ucode* object files specified on the command line, the runtime startup routine, and all of the runtime libraries are *ucode* linked. Optimization is done on the resulting *ucode* linked file and then it is linked as normal producing an ''a.out'' file. No resulting '.o' file is left from the *ucode* linked result as in previous releases. **−c** cannot be specified with **−O3**.

−feedback file
Used with the **−cord** option to specify *file* to be used as a feedback file. This *file* is produced by *prof*(1) with its **−feedback** option from an execution of the program produced by *pixie*(1). Multiple feedback files may be provided as *−feedback*file1 *−feedback*file2... *−feedback*filen.

−cord Run the procedure rearranger, *cord*(1), on the resulting file after linking. The rearrangement is done to improve the caching and paging performance of the program's text. The output of *cord*(1) is left in the file specified by the **−o** *output* option or 'a.out' by default. At least one **−feedback** *file* must be specified.

−j Compile the specified source programs, and leave the *ucode* object file output in corresponding files suffixed with '.u'. Please note that this switch is non-standard and may not be supported across product lines.

−ko *output*
Name the output file created by the ucode loader as *output*. This file is not removed. If this file is compiled, the object file is left in a file whose name consists of *output* with the suffix changed to a '.o'. If *output* has no suffix, a '.o' suffix is appended to *output*. Please note that this switch is non-standard and may not be supported across product lines.

−k Pass options that start with a **−k** to the ucode loader. This option is used to specify ucode libraries (with **−kl***x*) and other ucode loader options. Please note that this switch is non-standard and may not be supported across product lines.

−S Compile the specified source programs and leave the symbolic assembly language output in corresponding files suffixed with '.s'. If the **−O3** option is used, then a single file, u.out.s is produced.

−M Run only the macro preprocessor on the named C programs, requesting it to generate Makefile dependencies and send the result to the standard output.

−P Run only the C macro preprocessor and put the result for each source file (by suffix convention, i.e., '.c' and '.s') in a corresponding '.i' file. The '.i' file has no '#' lines in it. This sets the **−cpp** option.

−E Run only the C macro preprocessor files (regardless of any suffix or not), and send the result to the standard output. This sets the **−cpp** option.

−o *output*
Name the final output file *output*. If this option is used, the file 'a.out' is undisturbed.

−D*name=def*
−D*name* Define the *name* to the C macro preprocessor, as if by '#define'. If no definition is given, the name is defined as "1".

−U*name* Remove any initial definition of *name*.

−I*dir* '#include' files whose names do not begin with '/' are always sought first in the directory of the *file* argument, then in directories specified in −I options, and finally in the standard directory (**/usr/include**).

−nostdinc
This option will cause '#include' files never to be searched for in the standard directory (**/usr/include**).

−I This option will cause '#include' files never to be searched for in the standard directory (**/usr/include**). This option will be phased out in a future release; **−nostdinc** should be used in its place.

−G *num* Specify the maximum size, in bytes, of a data item that is to be accessed from the global pointer. *Num* is assumed to be a decimal number. If *num* is zero, no data is accessed from the global pointer. The default value for *num* is 8 bytes. Data stored off of the global pointer can be accessed by the program quickly, but this space is limited. Large programs may overflow the space accessed by the global pointer at load time. If the loader gives the error message "Bad −G num value", recompile with **−G 0**. Please note that this switch is non-standard and may not be supported across product lines.

−v Print the passes as they execute with their arguments and their input and output files.

−V This is an obsolete and inaccurate way to print version numbers of the compiler passes. Use *file*(1) instead.

−cpp Run the C macro preprocessor on C and assembly source files before compiling. This is the default.

−nocpp Do not run the C macro preprocessor on C and assembly source files before compiling.

−acpp uses an ANSI C preprocessor instead of the traditional cpp. __STDC__ is not defined.

−Olimit *num*
Specify the maximum size, in basic blocks, of a routine that will be optimized by the global optimizer. If a routine has more that this number of basic blocks it will not be optimized and a message will be printed. An option specifying that the global optimizer is to be run (**−O** or **−O2**) must also be specified. This option does not apply to **−O3** because all routines must be optimized. *Num* is assumed to be a decimal number. The default value for *num* is 1000 basic blocks.

The following options are specific to *cc*:

−mp Enable the multiprocessing directives. (Power C only)

−pca Run the *pca*(1) preprocessor to automatically discover parallelism in the source code. This also enables the multiprocessing (**−mp**) directive. There are two optional arguments: **−pca list** will run *pca* and also produce a listing file with suffix *.l* explaining which loops were parallelized and if not, why not. **−pca keep** runs *pca* producing the listing file and also keeps the transformed multiprocessed C intermediate file in a file with suffix *.m.* (Power C only)

−signed Cause all *char* declarations to be *signed char* declarations, the default is to treat them as *unsigned char* declarations.

−volatile
Causes all variables to be treated as *volatile.*

−varargs
Prints warnings for lines that may require the <*varargs.h*> macros. Please note that this switch is non-standard and may not be supported across product lines.

−float Cause the compiler to never promote expressions of type *float* to type *double* . This option does not affect float function arguments, which are always promoted to double unless function prototypes are used.

−Zg (NOTE: This flag is obsolete and should not be used. Instead, either the graphics library (**−lgl**) or the shared graphics library (**−lgl_s**) and the math library (**−lm**) should specified explicitly on the link line.) Load the program with the special files and libraries necessary for IRIS graphics programs. When this option is used, the graphics library (**−lgl**) and the math library (**−lm**) are loaded. Special files must be loaded for using graphics with other languages. Hence, *cc* must be able to determine the combination of languages involved in the link step or the special files required for each language must be specified in the link step (see *f77*(1) and *pc*(1)).

−prototypes
adds prototype error and warning messages. By default warnings and errors caused by the change to function prototype form in the standard headers are suppressed. Code generation is not affected by the option.

Use of this option is recommended.

−noprototypes
removes prototype error and warning messages (this is the default). The last of any sequence of **−prototypes** and **−noprototypes** is the one that is effective.

−real_frameptr
Use a register other than the stack-pointer(sp) as the frame pointer for each function compiled with this option; function entry is slightly slower for each function compiled this way. A function manipulating the stack pointer to provide local dynamic memory which is automatically freed on function exit (such as *alloca*() in GNUemacs) can be called from a function compiled with this option. The *alloca*() routine must provide enough free space after the newly allocated stack area for future calls to use to pass arguments. A complete implementation of alloca will be in a future

release. Source files compiled with this may be freely mixed with files compiled without it.

–dollar Allows the dollar sign ($) as a character in C identifiers, including allowing it as the leading character of an identifier. The option is provided solely for compatibility purposes. The dollar sign is not a standard C identifier character and its use is **not recommended.**

The options described below primarily aid compiler development and are not generally used:

–H*c* Halt compiling after the pass specified by the character *c*, producing an intermediate file for the next pass. The *c* can be [**Kfjusmoca**]. It selects the compiler pass in the same way as the **–t** option. If this option is used, the symbol table file produced and used by the passes, is the last component of the source file with the suffix changed to '.T' and is not removed. Please note that this switch is non-standard and may not be supported across product lines.

–K Build and use intermediate file names with the last component of the source file's name replacing its suffix with the conventional suffix for the type of file (for example '.B' file for binary *ucode,* produced by the front end). These intermediate files are never removed even when a pass encounters a fatal error. Please note that this switch is non-standard and may not be supported across product lines.

–W*c,arg1[,arg2...]*
Pass the argument[s] *argi* to the compiler pass *c*. The *c* is one of [**pKfjusmocablyz**]. The c selects the compiler pass in the same way as the **–t** option.

The options **–t[hpKfjusmocablrnyz]**, **–h***path,* and **–B***string* select a name to use for a particular pass, startup routine, or standard library. These arguments are processed from left to right so their order is significant. When the **–B** option is encountered, the selection of names takes place using the last **–h** and **–t** options. Therefore, the **–B** option is always required when using **–h** or **–t**. Sets of these options can be used to select any combination of names.

Any of the **–p[01]** options and any of the **–g[0123]** options must precede all **–B** options because they can affect the location of runtimes and what runtimes are used.

–t[hpKfjusmocablrnyz]

Select the names. The names selected are those designated by the characters following the **–t** option according to the following table:

Name	**Character**
include	h (see Note below)
cpp	p
pca	K (Power C only)
ccom	f (see below)
ujoin	j
uld	u
usplit	s
umerge	m
uopt	o
ugen	c
as0	a
as1	b
ld	l
[m]crt[1n].o	r
libprof1.a	n
ftoc	y
cord	z

Note: although *cc* may be used to compile source files in such languages as *FORTRAN* and *Pascal*, only the name used for the front-end of the *C* compiler is selected by the **–tf** option.

–h*path* Use *path* rather than the directory where the name is normally found. Please note that this switch is non-standard and may not be supported across product lines.

–B*string*

Append *string* to all names specified by the **–t** option. If no **–t** option has been processed before the **–B**, the **–t** option is assumed to be "hpKfjusmocablrnyz". This list designates all names. If no **–t** argument has been processed before the **–B** then a **–B***string* is passed to the loader to use with its **–l***x* arguments.

Invoking the compiler with a name of the form **cc***string* has the same effect as using a **–B***string* option on the command line.

If the environment variable TMPDIR is set, the value is used as the directory to place any temporary files rather than the default **/tmp**.

Other arguments are assumed to be either loader options or *C*-compatible object files, typically produced by an earlier *cc* run, or perhaps libraries of *C*-compatible routines. These files, together with the results of any compilations specified, are loaded in the order given, producing an executable program with the default name **a.out**.

FILES

file.c	input file
file.o	object file
a.out	loaded output
/tmp/ctm*	temporaries
/usr/lib/cpp	C macro preprocessor
/usr/lib/acpp	ANSI C macro preprocessor
/usr/lib/pca	multiprocessing analyzer (Power C only)
/usr/lib/ccom	C front end
/usr/lib/ccom_mp	multiprocessing C front end (Power C only)
/usr/lib/ujoin	binary ucode and symbol table joiner
/usr/bin/uld	ucode loader
/usr/lib/usplit	binary ucode and symbol table splitter
/usr/lib/umerge	procedure intergrator
/usr/lib/uopt	optional global ucode optimizer
/usr/lib/ugen	code generator
/usr/lib/as0	symbolic to binary assembly language translator
/usr/lib/as1	binary assembly language assembler and reorganizer
/usr/lib/mcrt1.o	startup for profiling
/usr/lib/crt1.o	runtime startup
/usr/lib/crtn.o	runtime startup
/usr/lib/libc.a	standard library, see *intro*(3)
/usr/lib/libc_s.a	shared standard library
/usr/lib/libfpe.a	floating point exception handler library, see *sigfpe*(3c)
/usr/lib/libgl.a	IRIS-4D graphics library
/usr/lib/libgl_s.a	IRIS-4D shared graphics library
/usr/lib/libm.a	math library
/usr/lib/libprof1.a	level 1 profiling library
/usr/lib/libc_mp.a	multiprocessing library (Power C only)
/usr/include	standard directory for '#include' files
/usr/bin/ld	MIPS loader
mon.out	file produced for analysis by *prof*(1)
/usr/lib/cord	procedure rearranger
/usr/lib/ftoc	feedback file to reorder list translator

SEE ALSO

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978
B. W. Kernighan, *Programming in C–a tutorial*
D. M. Ritchie, *C Reference Manual*
Power C User's Guide
as(1), cord(1), cpp(1), pca(1), dbx(1), edge(1), f77(1), ftoc(1), ld(1), pc(1), pixie(1), prof(1), what(1), sigfpe(3c), monstartup(3c).

DIAGNOSTICS

The diagnostics produced by *cc* are intended to be self-explanatory. Occasional messages may be produced by the assembler or loader.

BUGS

The compiler attempts to continue after finding semantic errors. These errors may result in compiler internal errors.

NOTES

The standard library, */usr/lib/libc.a*, is loaded by using the **–lc** loader option and not a full path name. The wrong one could be loaded if there are files with the name libc.a*string* in the directories specified with the **–L** loader option or in the default directories searched by the loader.

The handling of include directories and *libc.a* is confusing.

The shared standard library, */usr/lib/libc_s.a*, may be loaded instead of *libc.a* by specifying **–lc_s** as the last item on the link line.

The environment variables COMP_TARGET_ROOT, TOOLROOT, RLS_ID_OBJECT are used by *cc* while compiling the compiler itself. Their meaning is left undefined intentionally. They should **not** be set in your environment.

cc recognizes several switches (**–mp**, **–pca**, **–HK**, **–WK**, and **–tK**) related to multiprocessing. However; the associated programs and files are not present unless Power C has been installed.

Applications which provide their own *exit* function will not work with profiling (no profile data will be written out). One must use the *exit*(2) library function provided by *cc -p* when profiling.

NAME

cd – change working directory

SYNOPSIS

cd [directory]

DESCRIPTION

If *directory* is not specified, the value of shell parameter **$HOME** is used as the new working directory. If *directory* specifies a complete path starting with /, ., .., *directory* becomes the new working directory. If neither case applies, *cd* tries to find the designated directory relative to one of the paths specified by the **$CDPATH** shell variable. **$CDPATH** has the same syntax as, and similar semantics to, the **$PATH** shell variable. *cd* must have execute (search) permission in *directory*.

Because a new process is created to execute each command, *cd* would be ineffective if it were written as a normal command; therefore, it is recognized and is internal to the shell.

SEE ALSO

pwd(1), sh(1).
chdir(2) in the *Programmer's Reference Manual.*

NAME

cdc – change the delta commentary of an SCCS delta

SYNOPSIS

cdc −rSID [**−m**[mrlist]] [**−y**[comment]] files

DESCRIPTION

cdc changes the *delta commentary*, for the SID (SCCS IDentification string) specified by the **−r** keyletter, of each named SCCS file.

Delta commentary is defined to be the Modification Request (MR) and comment information normally specified via the *delta*(1) command (**−m** and **−y** keyletters).

If a directory is named, *cdc* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read (see *WARNINGS*) and each line of the standard input is taken to be the name of an SCCS file to be processed.

Arguments to *cdc*, which may appear in any order, consist of *keyletter* arguments and file names.

All the described *keyletter* arguments apply independently to each named file:

−r*SID* Used to specify the *S*CCS *ID*entification (SID) string of a delta for which the delta commentary is to be changed.

−m*mrlist* If the SCCS file has the **v** flag set [see *admin*(1)] then a list of MR numbers to be added and/or deleted in the delta commentary of the SID specified by the **−r** keyletter *may* be supplied. A null MR list has no effect.

MR entries are added to the list of MRs in the same manner as that of *delta*(1). In order to delete an MR, precede the MR number with the character **!** (see *EXAMPLES*). If the MR to be deleted is currently in the list of MRs, it is removed and changed into a "comment" line. A list of all deleted MRs is placed in the comment section of the delta commentary and preceded by a comment line stating that they were deleted.

If –**m** is not used and the standard input is a terminal, the prompt **MRs?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The **MRs?** prompt always precedes the **comments?** prompt (see –y keyletter).

MRs in a list are separated by blanks and/or tab characters. An unescaped new-line character terminates the MR list.

Note that if the v flag has a value [see *admin*(1)], it is taken to be the name of a program (or shell procedure) which validates the correctness of the MR numbers. If a non-zero exit status is returned from the MR number validation program, *cdc* terminates and the delta commentary remains unchanged.

–y*[comment]* Arbitrary text used to replace the *comment*(s) already existing for the delta specified by the –**r** keyletter. The previous comments are kept and preceded by a comment line stating that they were changed. A null *comment* has no effect.

If –y is not specified and the standard input is a terminal, the prompt **comments?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped new-line character terminates the *comment* text.

Simply stated, the keyletter arguments are either (1) if you made the delta, you can change its delta commentary; or (2) if you own the file and directory you can modify the delta commentary.

EXAMPLES

```
cdc –r1.6 –m"bl78-12345 !bl77-54321 bl79-00001" –ytrouble s.file
```

adds bl78-12345 and bl79-00001 to the MR list, removes bl77-54321 from the MR list, and adds the comment **trouble** to delta 1.6 of s.file.

```
cdc –r1.6 s.file
MRs? !bl77-54321 bl78-12345 bl79-00001
comments? trouble
```

does the same thing.

WARNINGS

If SCCS file names are supplied to the *cdc* command via the standard input (– on the command line), then the **–m** and **–y** keyletters must also be used.

FILES

x-file	[see *delta*(1)]
z-file	[see *delta*(1)]

SEE ALSO

admin(1), delta(1), get(1), prs(1), sccsfile(4).
help(1) in the *User's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

cflow – generate C flowgraph

SYNOPSIS

cflow [**–r**] [**–ix**] [**–i_**] [**–dnum**] files

DESCRIPTION

The *cflow* command analyzes a collection of C, yacc, lex, assembler, and object files and attempts to build a graph charting the external references. Files suffixed with **.y**, **.l**, and **.c** are yacced, lexed, and C-preprocessed as appropriate. The results of the preprocessed files, and files suffixed with **.i**, are then run through the first pass of *lint*(1). Files suffixed with **.s** are assembled. Assembled files, and files suffixed with **.o**, have information extracted from their symbol tables. The results are collected and turned into a graph of external references which is displayed upon the standard output.

Each line of output begins with a reference number, followed by a suitable number of tabs indicating the level, then the name of the global symbol followed by a colon and its definition. Normally only function names that do not begin with an underscore are listed (see the **–i** options below). For information extracted from C source, the definition consists of an abstract type declaration (e.g., **char ***), and, delimited by angle brackets, the name of the source file and the line number where the definition was found. Definitions extracted from object files indicate type information (if possible), followed by the name of the file in which the symbol appeared enclosed in angle brackets. Leading underscores in C-style external names are deleted.

Once a definition of a name has been printed, subsequent references to that name contain only the reference number of the line where the definition may be found. For undefined references, only < > is printed.

As an example, given the following in *file.c*:

```
int     i;

main()
{
        f();
        g();
        f();
}

f()
{
        i = h();
}
```

the command

```
cflow −ix file.c
```

produces the output

```
1       main: int(), <file.c 4>
2               f: int(), <file.c 11>
3                       h: <>
4                       i: int, <file.c 1>
5               g: <>
```

When the nesting level becomes too deep, the output of *cflow* can be piped to *pr*(1), using the **−e** option, to compress the tab expansion to something less than every eight spaces.

In addition to the **−D**, **−I**, and **−U** options [which are interpreted just as they are by *cc*(1) and *cpp*(1)], the following options are interpreted by *cflow*:

−r Reverse the "caller:callee" relationship producing an inverted listing showing the callers of each function. The listing is also sorted in lexicographical order by callee.

−ix Include external and static data symbols. The default is to include only functions in the flowgraph.

−i_ Include names that begin with an underscore. The default is to exclude these functions (and data if *−ix* is used).

−dnum The *num* decimal integer indicates the depth at which the flowgraph is cut off. By default this is a very large number. Attempts to set the cutoff depth to a nonpositive integer will be ignored.

DIAGNOSTICS

Complains about bad options. Complains about multiple definitions and only believes the first. Other messages may come from the various programs used (e.g., the C-preprocessor).

FILES

/usr/bin/cflow
/usr/lib/dag
/usr/lib/lpfx
/usr/lib/nmf
/usr/lib/flip
/usr/lib/lint1

SEE ALSO

as(1), cc(1), cpp(1), lex(1), lint(1), nm(1), yacc(1).
pr(1) in the *User's Reference Manual.*

BUGS

Cflow's analysis of object files is of questionable value. Best results are obtained when giving *cflow* source files whenever possible.

Files produced by *lex*(1) and *yacc*(1) cause the reordering of line number declarations which can confuse *cflow*. To get proper results, feed *cflow* the *yacc* or *lex* input.

NAME

check – check RCS status of a file

SYNOPSIS

check [-d] [-l] [*rcsdir* **]**

DESCRIPTION

check displays RCS checkout information for files in the current directory. This includes the revision level, checkout status of the file, and who the file is checked out to. *check* assumes *./RCS* is the default RCS directory. A different RCS directory can be specified with the *rcsdir* argument.

The **-d** switch causes *check* to display the date when the file was checked out in addition to the usual information. The **-l** switch causes *check* to display only the names of files currently checked out.

If *check* finds no files checked out, then it exits with status 0. If some files are checked out, it exits with status 1. If an error occurs, it exits with status 2.

SEE ALSO

ci(1), co(1), ident(1), merge(1), rcs(1), rcsdiff(1), rcsmerge(1), rlog(1), rcsfile(4).

NAME

chmod – change the permissions mode of a file or directory

SYNOPSIS

chmod mode file ...

chmod mode directory ...

DESCRIPTION

The permissions of the named *files* or *directories* are changed according to mode, which may be symbolic or absolute. Absolute changes to permissions are stated using octal numbers:

chmod *nnn file(s)*

where *n* is a number from 0 to 7. Symbolic changes are stated using mnemonic characters:

chmod *a operator b file(s)*

where *a* is one or more characters corresponding to **user**, **group**, or **other**; where *operator* is +, –, and =, signifying assignment of permissions; and where *b* is one or more characters corresponding to type of permission.

An absolute mode is given as an octal number constructed from the OR of the following modes:

04000	set user ID on execution
020#0	set group ID on execution if # is **7**, **5**, **3**, or **1**
	enable mandatory locking if # is **6**, **4**, **2**, or **0**
02000	set group inheritance (directories only)
01000	sticky bit (see discussion below)
0400	read by owner
0200	write by owner
0100	execute (search in directory) by owner
0070	read, write, execute (search) by group
0007	read, write, execute (search) by others

Symbolic changes are stated using letters that correspond both to access classes and to the individual permissions themselves. Permissions to a file may vary depending on your user identification number (UID) or group identification number (GID). Permissions are described in three sequences each having three characters:

User	Group	Other
rwx	rwx	rwx

This example (meaning that user, group, and others all have reading, writing, and execution permission to a given file) demonstrates two categories for granting permissions: the access class and the permissions themselves.

Thus, to change the mode of a file's (or directory's) permissions using *chmod*'s symbolic method, use the following syntax for mode:

[*who*] *operator* [*permission(s)*], ...

A command line using the symbolic method would appear as follows:

chmod g+rw *file(s)*

This command would make *file* readable and writable by the group.

The *who* part can be stated as one or more of the following letters:

u	user's permissions
g	group's permissions
o	others permissions

The letter **a** (all) is equivalent to **ugo** and is the default if *who* is omitted.

Operator can be + to add *permission* to the file's mode, – to take away *permission*, or = to assign *permission* absolutely. (Unlike other symbolic operations, = has an absolute effect in that it resets all other bits.) Omitting *permission* is only useful with = to take away all permissions.

Permission is any compatible combination of the following letters:

r	reading permission
w	writing permission
x	execution permission
s	user or group set-ID is turned on
t	sticky bit is turned on
l	mandatory locking will occur during access

Multiple symbolic modes separated by commas may be given, though no spaces may intervene between these modes. Operations are performed in the order given. Multiple symbolic letters following a single operator cause the corresponding operations to be performed simultaneously. The letter **s** is only meaningful with **u** or **g**, and **t** only works with **u**.

Mandatory file and record locking (**l**) refers to a file's ability to have its reading or writing permissions locked while a program is accessing that file. It is not possible to permit group execution and enable a file to be locked on execution at the same time. In addition, it is not possible to turn on the set-group-ID and enable a file to be locked on execution at the same time. The following examples,

chmod g+x,+l *file(s)*

chmod g+s,+l *file(s)*

are, therefore, illegal usages and will elicit error messages.

Only the owner of a file or directory (or the super-user) may change a file's mode. In order to turn on a file's set-group-ID, your own group ID must correspond to the file's, and group execution must be set.

The sticky bit is so named for historical reasons. It currently has no effect on any regular file. If a directory is writable and has the sticky bit set, files within that directory can be removed only if one or more of the following are true [see *unlink*(2)]:

the user owns the file
the user owns the directory
the file is writable by the user
the user is the super-user.

If the set–group–ID flag is set on a directory, then files created in that directory will have their group ID set to the group ID of the directory, otherwise the group ID of the file is set to the effective group ID of the creating process [see *chmod*(2)]. *mount*(1) provides an alternate way to set this behaviour for an entire file system [see *mount*(1) and *fstab*(4)].

EXAMPLES

chmod a–x *file*

chmod 444 *file*

The first examples deny execution permission to all. The absolute (octal) example permits only reading permissions.

chmod go+rw *file*

chmod 606 *file*

These examples make a file readable and writable by the group and others.

chmod +l *file*

This causes a file to be locked during access.

chmod =rwx,g+s *file*

chmod 2777 *file*

These last two examples enable all to read, write, and execute the file; and they turn on the set group-ID.

SEE ALSO

ls(1), mount(1).
chmod(2), unlink(2) in the *Programmer's Reference Manual.*

NAME

chown, chgrp – change owner or group

SYNOPSIS

chown [**–Rf**] owner[.group] file ...

chgrp [**–Rf**] group file ...

DESCRIPTION

Chown changes the owner of the *files* to *owner*. The owner may be either a decimal user ID or a login name found in the password file. An optional group may also be specified. The group may be either a decimal group ID or a group name found in the group file.

Chgrp changes the group ID of the *files* to *group*. The group may be either a decimal group ID or a group name found in the group file.

No errors, except for usage errors, are reported when the **–f** (force) option is given.

When the **–R** option is given, the command recursively descends its directory arguments setting the specified owner or group. When symbolic links are encountered, their ownership or group is changed, but they are not traversed.

If either command is invoked by other than the super-user, the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000 respectively, will be cleared.

Only the owner of a file (or the super-user) may change the owner or group of that file.

FILES

/etc/passwd
/etc/group

SEE ALSO

chmod(1).
chown(2), group(4), passwd(4) in the *Programmer's Reference Manual.*

NAME

ci – check in RCS revisions

SYNOPSIS

ci [options] file ...

DESCRIPTION

ci stores new revisions into RCS files. Each file name ending in ',v' is taken to be an RCS file, all others are assumed to be working files containing new revisions. *Ci* deposits the contents of each working file into the corresponding RCS file.

Pairs of RCS files and working files may be specified in 3 ways (see also the example section of *co*(1)).

1) Both the RCS file and the working file are given. The RCS file name is of the form *path1/workfile*,v and the working file name is of the form *path2/workfile*, where *path1/* and *path2/* are (possibly different or empty) paths and *workfile* is a file name.

2) Only the RCS file is given. Then the working file is assumed to be in the current directory and its name is derived from the name of the RCS file by removing *path1/* and the suffix ',v'.

3) Only the working file is given. Then the name of the RCS file is derived from the name of the working file by removing *path2/* and appending the suffix ',v'.

If the RCS file is omitted or specified without a path, then *ci* looks for the RCS file first in the directory ./RCS and then in the current directory.

For *ci* to work, the caller's login must be on the access list, except if the access list is empty or the caller is the superuser or the owner of the file. To append a new revision to an existing branch, the tip revision on that branch must be locked by the caller. Otherwise, only a new branch can be created. This restriction is not enforced for the owner of the file, unless locking is set to *strict* (see *rcs*(1)). A lock held by someone else may be broken with the *rcs* command.

Normally, *ci* checks whether the revision to be deposited is different from the preceding one. If it is not different, *ci* either aborts the deposit (if **–q** is given) or asks whether to abort (if **–q** is omitted). A deposit can be forced with the **–f** option.

For each revision deposited, *ci* prompts for a log message. The log message should summarize the change and must be terminated with a line containing a single '.' or a control-D. Log messages are limited to 16384 characters. If several files are checked in, *ci* asks whether to reuse the previous log message. If the standard input is not a terminal, *ci* suppresses the prompt and uses the same log message for all files. See also **–m**.

The number of the deposited revision can be given by any of the options **−r**, **−f**, **−k**, **−l**, **−u**, **−q** (see **−r**), or **−M**.

If the RCS file does not exist, *ci* creates it and deposits the contents of the working file as the initial revision (default number: 1.1). The access list is initialized to empty. Instead of the log message, *ci* requests descriptive text (see **−t** below).

−r[*rev*] assigns the revision number *rev* to the checked-in revision, releases the corresponding lock, and deletes the working file. This is also the default.

If *rev* is omitted, *ci* derives the new revision number from the caller's last lock. If the caller has locked the tip revision of a branch, the new revision is appended to that branch. The new revision number is obtained by incrementing the tip revision number. If the caller locked a non-tip revision, a new branch is started at that revision by incrementing the highest branch number at that revision. The default initial branch and level numbers are 1. If the caller holds no lock, but he is the owner of the file and locking is not set to *strict*, then the revision is appended to the trunk.

If *rev* indicates a revision number, it must be higher than the latest one on the branch to which *rev* belongs, or must start a new branch.

If *rev* indicates a branch instead of a revision, the new revision is appended to that branch. The level number is obtained by incrementing the tip revision number of that branch. If *rev* indicates a non-existing branch, that branch is created with the initial revision numbered *rev.1*.

Exception: On the trunk, revisions can be appended to the end, but not inserted.

−f[*rev*] forces a deposit; the new revision is deposited even it is not different from the preceding one.

−k[*rev*] searches the working file for keyword values to determine its revision number, creation date, author, and state (see *co* (1)), and assigns these values to the deposited revision, rather than computing them locally. A revision number given by a command option overrides the number in the working file. This option is useful for software distribution. A revision that is sent to several sites should be checked in with the **−k** option at these sites to preserve its original number, date, author, and state.

−l[*rev*]	works like **−r**, except it performs an additional *co* **−l** for the deposited revision. Thus, the deposited revision is immediately checked out again and locked. This is useful for saving a revision although one wants to continue editing it after the checkin.
−u[*rev*]	works like **−l**, except that the deposited revision is not locked. This is useful if one wants to process (e.g., compile) the revision immediately after checkin.
−q[*rev*]	quiet mode; diagnostic output is not printed. A revision that is not different from the preceding one is not deposited, unless **−f** is given.
−M[*rev*]	use the files last-modified date for the revision instead of the current date. If the **-l** or **-u** options are also given, the file's last-modified date will be unchanged.
−m*msg*	uses the string *msg* as the log message for all revisions checked in.
−n*name*	assigns the symbolic name *name* to the number of the checked-in revision. *ci* prints an error message if *name* is already assigned to another number.
−N*name*	same as **−n**, except that it overrides a previous assignment of *name*.
−s*state*	sets the state of the checked-in revision to the identifier *state*. The default is *Exp*.
−t[*txtfile*]	writes descriptive text into the RCS file (deletes the existing text). If *txtfile* is omitted, *ci* prompts the user for text supplied from the standard input, terminated with a line containing a single '.' or CTRL-d. Otherwise, the descriptive text is copied from the file *txtfile*. During initialization, descriptive text is requested even if **−t** is not given. The prompt is suppressed if standard input is not a terminal.

DIAGNOSTICS

For each revision, *ci* prints the RCS file, the working file, and the number of both the deposited and the preceding revision. The exit status always refers to the last file checked in, and is 0 if the operation was successful, 1 otherwise.

FILE MODES

An RCS file created by *ci* inherits the read and execute permissions from the working file. If the RCS file exists already, *ci* preserves its read and execute permissions. *ci* always turns off all write permissions of RCS files.

FILES

The caller of the command must have read/write permission for the directories containing the RCS file and the working file, and read permission for the RCS file itself. A number of temporary files are created. A semaphore file is created in the directory containing the RCS file. *ci* always creates a new RCS file and unlinks the old one. This strategy makes links to RCS files useless.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.1 ; Release Date: 83/04/04.
Copyright © 1982 by Walter F. Tichy.

SEE ALSO

co(1), ident(1), rcs(1), rcsdiff(1), rcsintro(1), rcsmerge(1), rlog(1), rcsfile(4)
Walter F. Tichy, ''Design, Implementation, and Evaluation of a Revision Control System,'' in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

NAME

clear – clear terminal screen

SYNOPSIS

clear

DESCRIPTION

Clear clears your screen if this is possible. It looks in the environment for the terminal type and then uses the *terminfo*(4) database to figure out how to clear the screen.

FILES

/usr/lib/terminfo/?/*	compiled terminal description database

SEE ALSO

terminfo(4)

NAME

cmp – compare two files

SYNOPSIS

cmp [–l] [–s] file1 file2

DESCRIPTION

The two files are compared. (If *file1* is –, the standard input is used.) Under default options, *cmp* makes no comment if the files are the same; if they differ, it announces the byte and line number at which the difference occurred. If one file is an initial subsequence of the other, that fact is noted.

Options:

–l	Print the byte number (decimal) and the differing bytes (octal) for each difference.
–s	Print nothing for differing files; return codes only.

SEE ALSO

comm(1), diff(1).

DIAGNOSTICS

Exit code 0 is returned for identical files, 1 for different files, and 2 for an inaccessible or missing argument.

NAME

co – check out RCS revisions

SYNOPSIS

co [options] file ...

DESCRIPTION

co retrieves revisions from RCS files. Each file name ending in ',v' is taken to be an RCS file. All other files are assumed to be working files. *co* retrieves a revision from each RCS file and stores it into the corresponding working file.

Pairs of RCS files and working files may be specified in 3 ways (see also the example section).

1) Both the RCS file and the working file are given. The RCS file name is of the form *path1/workfile*,v and the working file name is of the form *path2/workfile*, where *path1/* and *path2/* are (possibly different or empty) paths and *workfile* is a file name.

2) Only the RCS file is given. Then the working file is created in the current directory and its name is derived from the name of the RCS file by removing *path1/* and the suffix ',v'.

3) Only the working file is given. Then the name of the RCS file is derived from the name of the working file by removing *path2/* and appending the suffix ',v'.

If the RCS file is omitted or specified without a path, then *co* looks for the RCS file first in the directory ./RCS and then in the current directory.

Revisions of an RCS file may be checked out locked or unlocked. Locking a revision prevents overlapping updates. A revision checked out for reading or processing (e.g., compiling) need not be locked. A revision checked out for editing and later checkin must normally be locked. Locking a revision currently locked by another user fails. (A lock may be broken with the *rcs* (1) command.) *co* with locking requires the caller to be on the access list of the RCS file, unless he is the owner of the file or the superuser, or the access list is empty. *co* without locking is not subject to accesslist restrictions.

A revision is selected by number, checkin date/time, author, or state. If none of these options are specified, the latest revision on the trunk is retrieved. When the options are applied in combination, the latest revision that satisfies all of them is retrieved. The options for date/time, author, and state retrieve a revision on the *selected branch.* The selected branch is either derived from the revision number (if given), or is the highest branch on the trunk. A revision number may be attached to one of the options **–l**, **–p**, **–q**, **–M**, or **–r**.

A *co* command applied to an RCS file with no revisions creates a zero-length file. *co* always performs keyword substitution (see below).

−l[*rev*] locks the checked out revision for the caller. If omitted, the checked out revision is not locked. See option −r for handling of the revision number *rev*.

−p[*rev*] prints the retrieved revision on the std. output rather than storing it in the working file. This option is useful when *co* is part of a pipe.

−q[*rev*] quiet mode; diagnostics are not printed.

−M[*rev*] the checked out file's last-modified date is set to the revision date instead of the current date.

−d*date* retrieves the latest revision on the selected branch whose checkin date/time is less than or equal to *date*. The date and time may be given in free format and are converted to local time. Examples of formats for *date*:

22-April-1982, 17:20-CDT,
2:25 AM, Dec. 29, 1983,
Tue-PDT, 1981, 4pm Jul 21 (free format),
Fri, April 16 15:52:25 EST 1982 (output of ctime).

Most fields in the date and time may be defaulted. *co* determines the defaults in the order year, month, day, hour, minute, and second (most to least significant). At least one of these fields must be provided. For omitted fields that are of higher significance than the highest provided field, the current values are assumed. For all other omitted fields, the lowest possible values are assumed. For example, the date "20, 10:30" defaults to 10:30:00 of the 20th of the current month and current year. The date/time must be quoted if it contains spaces.

−r[*rev*] retrieves the latest revision whose number is less than or equal to *rev*. If *rev* indicates a branch rather than a revision, the latest revision on that branch is retrieved. *Rev* is composed of one or more numeric or symbolic fields separated by '.'. The numeric equivalent of a symbolic field is specified with the −n option of the commands *ci* and *rcs*.

−s*state* retrieves the latest revision on the selected branch whose state is set to *state*.

–w[*login*]	retrieves the latest revision on the selected branch which was checked in by the user with login name *login.* If the argument *login* is omitted, the caller's login is assumed.
–j*joinlist*	generates a new revision which is the join of the revisions on *joinlist. Joinlist* is a comma-separated list of pairs of the form *rev2:rev3*, where *rev2* and *rev3* are (symbolic or numeric) revision numbers. For the initial such pair, *rev1* denotes the revision selected by the options –l, ..., –w. For all other pairs, *rev1* denotes the revision generated by the previous pair. (Thus, the output of one join becomes the input to the next.)

For each pair, *co* joins revisions *rev1* and *rev3* with respect to *rev2*. This means that all changes that transform *rev2* into *rev1* are applied to a copy of *rev3*. This is particularly useful if *rev1* and *rev3* are the ends of two branches that have *rev2* as a common ancestor. If *rev1* < *rev2* < *rev3* on the same branch, joining generates a new revision which is like *rev3*, but with all changes that lead from *rev1* to *rev2* undone. If changes from *rev2* to *rev1* overlap with changes from *rev2* to *rev3*, *co* prints a warning and includes the overlapping sections, delimited by the lines <<<<<<< *rev1*, =======, and >>>>>>> *rev3*.

For the initial pair, *rev2* may be omitted. The default is the common ancestor. If any of the arguments indicate branches, the latest revisions on those branches are assumed. If the option –l is present, the initial *rev1* is locked.

KEYWORD SUBSTITUTION

Strings of the form *$keyword$* and *$keyword:...$* embedded in the text are replaced with strings of the form *$keyword: value $*, where *keyword* and *value* are pairs listed below. Keywords may be embedded in literal strings or comments to identify a revision.

Initially, the user enters strings of the form *$keyword$*. On checkout, *co* replaces these strings with strings of the form *$keyword: value $*. If a revision containing strings of the latter form is checked back in, the value fields will be replaced during the next checkout. Thus, the keyword values are automatically updated on checkout.

Keywords and their corresponding values:

$Author$	The login name of the user who checked in the revision.
$Class$	Prog, Def, Doc, or Test, depending on the class assigned to the file with the -c option of the *rcs* command.

$Date$	The date and time the revision was checked in.
$Header$	A standard header containing the RCS file name, the revision number, the date, the author, and the state.
$Locker$	The login name of the user who locked the revision (empty if not locked).
Log	The log message supplied during checkin, preceded by a header containing the RCS file name, the revision number, the author, and the date. Existing log messages are NOT replaced. Instead, the new log message is inserted after *$Log:...$*. This is useful for accumulating a complete change log in a source file.
$Revision$	The revision number assigned to the revision.
$Source$	The full pathname of the RCS file.
$State$	The state assigned to the revision with *rcs* –s or *ci* –s.

DIAGNOSTICS

The RCS file name, the working file name, and the revision number retrieved are written to the diagnostic output. The exit status always refers to the last file checked out, and is 0 if the operation was successful, 1 otherwise.

EXAMPLES

Suppose the current directory contains a subdirectory 'RCS' with an RCS file 'io.c,v'. Then all of the following commands retrieve the latest revision from 'RCS/io.c,v' and store it into 'io.c'.

```
co io.c;  co RCS/io.c,v;  co io.c,v;
co io.c RCS/io.c,v;  co io.c io.c,v;
co RCS/io.c,v io.c;  co io.c,v io.c;
```

FILE MODES

The working file inherits the read and execute permissions from the RCS file. In addition, the owner write permission is turned on, unless the file is checked out unlocked and locking is set to *strict* (see *rcs* (1)).

If a file with the name of the working file exists already and has write permission, *co* aborts the checkout if –q is given, or asks whether to abort if –q is not given. If the existing working file is not writable, it is deleted before the checkout.

FILES

The caller of the command must have write permission in the working directory, read permission for the RCS file, and either read permission (for reading) or read/write permission (for locking) in the directory which contains the RCS file.

A number of temporary files are created. A semaphore file is created in the directory of the RCS file to prevent simultaneous update.

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.1 ; Release Date: 83/04/04.
Copyright © 1982 by Walter F. Tichy.

SEE ALSO

check(1), ci(1), ident(1), rcs(1), rcsdiff(1), rcsintro(1), rcsmerge(1), rlog(1), rcsfile(4)
Walter F. Tichy, ''Design, Implementation, and Evaluation of a Revision Control System,'' in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

LIMITATIONS

The option –d gets confused in some circumstances, and accepts no date before 1970. There is no way to suppress the expansion of keywords, except by writing them differently. In nroff and troff, this is done by embedding the null-character '\&' into the keyword.

BUGS

The option –j does not work for files that contain lines with a single '.'.

NAME

col – filter reverse line-feeds

SYNOPSIS

col [**–b**] [**–f**] [**–x**] [**–p**]

DESCRIPTION

col reads from the standard input and writes onto the standard output. It performs the line overlays implied by reverse line feeds (ASCII code ESC-**7**), and by forward and reverse half-line-feeds (ESC-**9** and ESC-**8**). *col* is particularly useful for filtering multicolumn output made with the **.rt** command of *nroff* and output resulting from use of the *tbl*(1) preprocessor.

If the **–b** option is given, *col* assumes that the output device in use is not capable of backspacing. In this case, if two or more characters are to appear in the same place, only the last one read will be output.

Although *col* accepts half-line motions in its input, it normally does not emit them on output. Instead, text that would appear between lines is moved to the next lower full-line boundary. This treatment can be suppressed by the **–f** (fine) option; in this case, the output from *col* may contain forward half-line-feeds (ESC-**9**), but will still never contain either kind of reverse line motion.

Unless the **–x** option is given, *col* will convert white space to tabs on output wherever possible to shorten printing time.

The ASCII control characters SO (\017) and SI (\016) are assumed by *col* to start and end text in an alternate character set. The character set to which each input character belongs is remembered, and on output SI and SO characters are generated as appropriate to ensure that each character is printed in the correct character set.

On input, the only control characters accepted are space, backspace, tab, return, new-line, SI, SO, VT (\013), and ESC followed by **7**, **8**, or **9**. The VT character is an alternate form of full reverse line-feed, included for compatibility with some earlier programs of this type. All other non-printing characters are ignored.

Normally, *col* will ignore any escape sequences unknown to it that are found in its input; the **–p** option may be used to cause *col* to output these sequences as regular characters, subject to overprinting from reverse line motions. The use of this option is highly discouraged unless the user is fully aware of the textual position of the escape sequences.

SEE ALSO

nroff(1), tbl(1) in the *DOCUMENTER's WORKBENCH Software Release 2.0 Technical Discussion and Reference Manual*.

NOTES

The input format accepted by *col* matches the output produced by *nroff* with either the **–T37** or **–Tlp** options. Use **–T37** (and the **–f** option of *col*) if the ultimate disposition of the output of *col* will be a device that can interpret half-line motions, and **–Tlp** otherwise.

BUGS

Cannot back up more than 128 lines.
Allows at most 800 characters, including backspaces, on a line.
Local vertical motions that would result in backing up over the first line of the document are ignored. As a result, the first line must not have any superscripts.

NAME

collide – look for name collisions between libraries

SYNOPSIS

collide [–l] [–d] *file* ...

DESCRIPTION

When porting a large application a critical question is ''How do I know if names in this application conflict with names in libraries I must use?'' *Collide* answers that question by finding name collisions across archive libraries and object files.

Given a list of files, *collide* finds all duplicate external names and writes those names to standard output. There is no output if there are no conflicts.

The **–l** (longform) option adds the object file names to the output; these are occasionally interesting. This must be the first option to *collide* if it is specified at all.

The **–d** (show duplicates) option reports duplicate external names on standard error. This reports the same problems *ar* would if run on the same object files but in more detail. **–d** applies to object files, not archive libraries.

The *files* named may be either archive libraries or object files. A list of object files is gathered together (by */usr/lib/collide1*) just as is done by *ar*(1). Gathering of a list of objects stops when an archive name is seen. Object gathering is important because a *common* declaration may appear in multiple files yet it does not reflect a name duplication.

Collide is a shell script that uses */usr/lib/collide1* to extract the public symbols from each archive (or gathered collection of objects) and notes the object file name and archive name along with the symbol. You may find it useful to find which library or object contains a name. For example,

```
/usr/lib/collide1 /usr/lib/libc.a | grep strcpy
```

If there are too many objects to list them all on the command line, use the *ar*(1) command to build an archive.

Examples:

```
cd /usr/lib
collide libgl_s.a libc.a libcps.a /tmp/mylib.a
cd /usr/tmp
# a.o b.o treated as if in archive "none".
# c.o d.o treated as if in a separate archive "none".
collide –d a.o b.o /usr/lib/libc.a c.o d.o

ar cr mylib.a *.o
```

```
collide  mylib.a /usr/lib/libc.a

# Not useful. All the externals are collisions.
collide libc_s.a libc.a
```

FILES

/usr/sbin/collide
/usr/lib/collide1

NOTES

The format of a line written by *collide1* is:

```
variable-name objectfile-name archive-name
```

Each name is separated by a tab; the line ends with a newline. There are no blanks or extraneous characters in the output.

If you supply object names to *collide* it is a good idea to use the **–d** option.

While the C language does not have FORTRAN *common* declarations per se, the declaration `int c;` will, if it appears outside of any function declaration be marked in the symbol table as *common*.

BUGS

common declarations which appear in different archives can cause false reports of duplications. This should not happen if you put all your objects in a single archive or list all your objects together on the command line.

NAME

comb – combine SCCS deltas

SYNOPSIS

comb [**–o**][**–s**][**–p** sid][**–c** list] files

DESCRIPTION

comb generates a shell procedure [see *sh*(1)] which, when run, will reconstruct the given SCCS files. The reconstructed files will, hopefully, be smaller than the original files. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *comb* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the input is taken to be the name of an SCCS file to be processed; non-SCCS files and unreadable files are silently ignored. The generated shell procedure is written on the standard output.

The keyletter arguments are as follows. Each is explained as though only one named file is to be processed, but the effects of any keyletter argument apply independently to each named file.

–o For each **get –e** generated, this argument causes the reconstructed file to be accessed at the release of the delta to be created, otherwise the reconstructed file would be accessed at the most recent ancestor. Use of the **–o** keyletter may decrease the size of the reconstructed SCCS file. It may also alter the shape of the delta tree of the original file.

–s This argument causes *comb* to generate a shell procedure which, when run, will produce a report giving, for each file: the file name, size (in blocks) after combining, original size (also in blocks), and percentage change computed by:

100 * (original – combined) / original

It is recommended that before any SCCS files are actually combined, one should use this option to determine exactly how much space is saved by the combining process.

–pSID The *CCS* ID*entification* string (SID) of the oldest delta to be preserved. All older deltas are discarded in the reconstructed file.

–c list A

list (see *get*(1) for the syntax of a *list*) of deltas to be preserved. All other deltas are discarded.

If no keyletter arguments are specified, *comb* will preserve only leaf deltas and the minimal number of ancestors needed to preserve the tree.

FILES

s.COMB	The name of the reconstructed SCCS file.
comb?????	Temporary.

SEE ALSO

admin(1), delta(1), get(1), prs(1), sccsfile(4).
help(1), sh(1) in the *User's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

comb may rearrange the shape of the tree of deltas. It may not save any space; in fact, it is possible for the reconstructed file to actually be larger than the original.

NAME

comm – select or reject lines common to two sorted files

SYNOPSIS

comm [– [**123**]] file1 file2

DESCRIPTION

comm reads *file1* and *file2*, which should be ordered in ASCII collating sequence (see *sort*(1)), and produces a three-column output: lines only in *file1*; lines only in *file2*; and lines in both files. The file name – means the standard input.

Flags 1, 2, or 3 suppress printing of the corresponding column. Thus **comm –12** prints only the lines common to the two files; **comm –23** prints only lines in the first file but not in the second; **comm –123** prints nothing.

SEE ALSO

cmp(1), diff(1), sort(1), uniq(1).

NAME

compress, uncompress, zcat – compress and expand data

SYNOPSIS

compress [**–f**] [**–v**] [**–c**] [**–b** *bits*] [*name* ...]
uncompress [**–f**] [**–v**] [**–c**] [*name* ...]
zcat [*name* ...]

DESCRIPTION

Compress reduces the size of the named files using adaptive Lempel-Ziv coding. Whenever possible, each file is replaced by one with the extension **.Z**, while keeping the same ownership modes, access and modification times. If no files are specified, the standard input is compressed to the standard output. Compressed files can be restored to their original form using *uncompress* or *zcat.*

The **–f** option will force compression of *name*, even if it does not actually shrink or the corresponding *name*.Z file already exists. Except when run in the background under */bin/sh*, if **–f** is not given the user is prompted as to whether an existing *name*.Z file should be overwritten.

The **–c** (''cat'') option makes *compress/uncompress* write to the standard output; no files are changed. The nondestructive behavior of *zcat* is identical to that of *uncompress* **–c.**

Compress uses the modified Lempel-Ziv algorithm popularized in "A Technique for High Performance Data Compression", Terry A. Welch, *IEEE Computer,* vol. 17, no. 6 (June 1984), pp. 8-19. Common substrings in the file are first replaced by 9-bit codes 257 and up. When code 512 is reached, the algorithm switches to 10-bit codes and continues to use more bits until the limit specified by the **–b** flag is reached (default 16). *Bits* must be between 9 and 16. The default can be changed in the source to allow *compress* to be run on a smaller machine.

After the *bits* limit is attained, *compress* periodically checks the compression ratio. If it is increasing, *compress* continues to use the existing code dictionary. However, if the compression ratio decreases, *compress* discards the table of substrings and rebuilds it from scratch. This allows the algorithm to adapt to the next "block" of the file.

Note that the **–b** flag is omitted for *uncompress,* since the *bits* parameter specified during compression is encoded within the output, along with a magic number to ensure that neither decompression of random data nor recompression of compressed data is attempted.

The amount of compression obtained depends on the size of the input, the number of *bits* per code, and the distribution of common substrings. Typically, text such as source code or English is reduced by 50–60%. Compression is generally much better than that achieved by Huffman coding (as used in *pack*), or adaptive Huffman coding (*compact*), and takes less time to compute.

The −v option causes the printing of the percentage reduction of each file.

If an error occurs, exit status is 1, else if the last file was not compressed because it became larger, the status is 2; else the status is 0.

DIAGNOSTICS

Usage: compress [−fvc] [−b maxbits] [file ...]
: Invalid options were specified on the command line.

Missing maxbits
: Maxbits must follow −b.

file: not in compressed format
: The file specified to *uncompress* has not been compressed.

file: compressed with *xx* bits, can only handle *yy* bits
: *File* was compressed by a program that could deal with more *bits* than the compress code on this machine. Recompress the file with smaller *bits*.

file: already has .Z suffix -- no change
: The file is assumed to be already compressed. Rename the file and try again.

file: filename too long to tack on .Z
: The file cannot be compressed because its name is longer than 12 characters. Rename and try again. This message does not occur on BSD systems.

file already exists; do you wish to overwrite (y or n)?
: Respond "y" if you want the output file to be replaced; "n" if not.

uncompress: corrupt input
: A SIGSEGV violation was detected which usually means that the input file is corrupted.

Compression: *xx.xx%*
: Percentage of the input saved by compression. (Relevant only for −v.)

-- not a regular file: unchanged
: When the input file is not a regular file, (e.g. a directory), it is left unaltered.

-- has *xx* other links: unchanged
: The input file has links; it is left unchanged. See *ln*(1)

for more information.

-- file unchanged

No savings is achieved by compression. The input remains virgin.

BUGS

Although compressed files are compatible between machines with large memory, **–b**12 should be used for file transfer to architectures with a small process data space (64KB or less, as exhibited by the DEC PDP series, the Intel 80286, etc.)

compress should be more flexible about the existence of the '.Z' suffix.

NAME

confirm – display a message in a window and request a response

SYNOPSIS

confirm [**–b** button-name ...] [**–t** "title-string" ...]

DESCRIPTION

confirm displays a window containing a line of test for each **–t** argument specified, and a button for each **–b** argument specified. When one of the buttons is pressed, the label of that button is written to *confirm's* standard output. This allows shell scripts to ask questions.

EXAMPLE

The following shell script will display a window, asking the user a yes or no question.

```
#! /bin/sh
case `confirm -t "Really power down the computer?" -b No -b Yes`
in
   Yes) shutdown ;;
   No) ;;
esac
```

BUGS

There can be at most three lines of titles specified with **–t** and three buttons specified with **–b.**
The window which appears is fixed size, and the text is not wrapped, so a lengthy message can be truncated.
The buttons are of fixed size, so lengthy button titles will not fit on the buttons.

SEE ALSO

inform(1G)

NAME

cord – rearranges procedures in an executable to facilitate better cache mapping.

SYNOPSIS

cord [**–w**] [**–v**] [**–o** outfile] [**–f**] [**–c** cachesize] [**–p** maxphases] obj reorder

DESCRIPTION

The *cord*(1) command may be used to rearrange procedures in an executable object to correspond with an ordering provided in a *reorder* file. Normally, the ordering is arranged either to maximize cache hit rate or to minimize paging. The reorder file may be generated either by hand or by the *ftoc*(1) program. *Ftoc*(1) produces the *reorder* file from one or more profile feedback files produced by the *-feedback* option to *prof* (see *prof*(1)).

The *reorder* file may contain three types of lines. Lines beginning with the character '*$*' are considered *directive* lines, which are commands to *cord*. The only directive currently understood is *$phase*. This directive will consider the rest of the file (until the end of file or next $phase) as a new phase of the program and will order the procedures accordingly. Procedures may appear in more than one phase, resulting in more than one copy of it in the final binary. *Cord* will try to relocate references to a procedure to a copy in the requesting phase's list of procedures first and then a random copy if one is not found. Lines beginning with either a blank or the character '#' are considered *comments*, and are ignored. Other lines are considered *procedure* lines, each of which disambiguates a procedure in the object being reordered. Procedure lines consist of the name of the source file containing the procedure, followed by a blank, followed by a procedure name. If the procedure name is followed by a blank, the remainder of the line is considered a comment. Names of nested procedures (e.g., Pascal) are qualified by indicating the nesting by *outermost.outer. ... innermost*. Thus, if file input.p contains a procedure *getline* which is local to procedure *getpage* which, in turn, is local to procedure *readinput*, a commented reorder line for *getline* could be written

input.p readinput.getpage.getline *local proc getline*

The facilities of *cord*(1) are most easily accessed by using the *-cord* option to one of the compiler drivers (e.g., *cc*(1)). Once you have obtained one or more profile feedback files for your program, simply relink your program with the *-cord* and *-feedback* options. For example, given a feedback file *foo.fd* for a program *foo*, you can produce a reordered version of *foo* by adding the options *-feedback foo.fd* and *–cord* to the *cc*(1) command in the link step. *Cord* options can be specified to *cc*(1) with the switch

-Wz,cordarg0,cordarg1, ... Since the new executable was reordered using feedback files from *prof*(1), any performance improvement will be limited in scope to those situations covered by the profile dataset(s).

Utilizing a hand-optimized *reorder* file necessitates running *cord*(1) by hand. First, produce a version of your executable with the relocation information intact by adding the switches *-z*, *-r*, and *-d* to the link command to *ld*(1). (These switches may be passed to *ld*(1) from *cc*(1) by adding the single *cc* switch *-Wl,-z,-d,-r*.) After linking, simply execute *cord*(1) as indicated above on the resultant file.

The *cord*(1) command accepts these options:

-v prints verbose information. This includes listing those procedures considered part of other procedures and which cannot be rearranged (these are basically assembler procedures that may contain relative branches to other procedures rather than relocatable ones). The listing also lists those procedures in the flipped area (if any) and a mapping of old locations to new. If you invoke *cord*(1) via *cc*(1) using the *-v* switch, the *-v* switch will be passed along to *cord*(1). The addition of multiple *-v* switches increases the verbosity of the output. The information provided by multiple *-v* switches is seldom informative to the general user.

-f flip the first cachepage size procedures. The assumption when cord was written was that procedures would be reordered by procedure density (cycles/byte). This option ensures that the densest part of each page following the first cachepage would conflict with the least dense part of the first cachepage.

–c *cachesize*
cachesize is a decimal constant which specifies the cache size of the machine you want to execute on in bytes. This only affects the *-f* option. If not specified 65536 is used.

–o *outputfile*
specifies the output file. If not specified *a.out* is used.

–p *phasemax*
specifies the maximum number of phases allowed. The default is 20.

−w	suppress warning messages.

FILES

/usr/lib/cord

BUGS

Although *cord*(1) updates the symbol and proc tables of the rearranged executable with correct function addresses, it is unable to update the linenumber tables, making debugging of the rearranged executable confusing at best. It is suggested that any needed debugging be performed prior to rearrangement with *cord*(1).

SEE ALSO

prof(1), cc(1), ftoc(1), ld(1), *MIPS Languages Programmer Guide*

NAME

cp, ln, mv – copy, link or move files

SYNOPSIS

cp [**–ir**] file1 [file2 ...] target
ln [**–sif**] file1 [file2 ...] target
mv [**–if**] file1 [file2 ...] target

DESCRIPTION

file1 is copied (linked, moved) to *target*. Under no circumstance can *file1* and *target* be the same (take care when using *sh*(1) metacharacters). If *target* is a directory, then one or more files are copied (linked, moved) to that directory. If *target* is a file, its contents are destroyed.

If *mv* or *ln* determines that the mode of *target* forbids writing, it will print the mode (see *chmod*(2)), ask for a response, and read the standard input for one line; if the line begins with **y**, the *mv* or *ln* occurs, if permissible; if not, the command exits. When the **–f** option is used or if the standard input is not a terminal, no questions are asked and the *mv* or *ln* is done.

Only *mv* will allow *file1* to be a directory, in which case the directory rename will occur only if the two directories have the same parent; *file1* is renamed *target*. If *file1* is a file and *target* is a link to another file with links, the other links remain and *target* becomes a new file.

When using *cp*, if *target* is not a file, a new file is created which has the same mode as *file1* except that the sticky bit is not set unless you are super-user; the owner and group of *target* are those of the user. If *target* is a file, copying a file into *target* does not change its mode, owner, nor group. The last modification time of *target* (and last access time, if *target* did not exist) and the last access time of *file1* are set to the time the copy was made. If *target* is a link to a file, all links remain and the file is changed.

With the **–s** option *ln* creates symbolic links. A symbolic link is a special kind of file whose contents are the name of another file (see *symlink*(2)). A symbolic link contains the name of the file to which it is linked. Most system calls, including *open*(2), *stat*(2), and *access*(2), substitute these contents when the name of a symbolic link occurs in a pathname. This process is known as ''following'' symbolic links. *stat*(2), *readlink*(2), *symlink*(2), and *unlink*(2) also do this substitution, except on the last component of the pathname. Thus, they are said to ''not follow'' symbolic links. Symbolic links may span file systems and may refer to directories. Note that *cp* ''follows'' symbolic links, while *mv* and *ln* do ''not follow'' symbolic links.

If the **–i** option is specified with *cp* or *mv* and *target* exists, the user is prompted with the message:

overwrite *target*?

If the user answers with a line starting with ‘y’, the move or copy continues. Any other reply prevents the command from completing.

If the −r option is specified with *cp* and any of the source files are directories, *cp* copies each subtree rooted at that name; in this case the target must be a directory.

SEE ALSO

chmod(1), cpio(1), rm(1).

WARNINGS

ln without −s will not link across file systems. This restriction is necessary because file systems can be added and removed.

BUGS

If *file1* and *target* lie on different file systems, *mv* must copy the file and delete the original. In this case any linking relationship with other files is lost.

NAME

cpio – copy file archives in and out

SYNOPSIS

cpio –o [**acBvV**] [**–C** *bufsize*] [[**–O** *file*] [**–M** *message*]]

cpio –i [**BcdmrtuvVfsSb6k**] [**–C** *bufsize*] [[**–I** *file*] [**–M** *message*]] [*pattern* ...]

cpio –p [**adlmuvV**] directory

DESCRIPTION

cpio –o (copy out) reads the standard input to obtain a list of path names and copies those files onto the standard output together with path name and status information. Output is padded to a 512-byte boundary by default.

cpio –i (copy in) extracts files from the standard input, which is assumed to be the product of a previous **cpio –o**. Only files with names that match *patterns* are selected. *patterns* are regular expressions given in the filename-generating notation of *sh*(1). In *patterns*, meta-characters **?**, *, and [. . .] match the slash (/) character, and backslash (\) is an escape character. A **!** meta-character means *not*. (For example, the **!abc*** pattern would exclude all files that begin with **abc**.) Multiple *patterns* may be specified and if no *patterns* are specified, the default for *patterns* is * (i.e., select all files). Each *pattern* must be enclosed in double quotes otherwise the name of a file in the current directory is used. Extracted files are conditionally created and copied into the current directory tree based upon the options described below. The permissions of the files will be those of the previous **cpio –o**. The owner and group of the files will be that of the current user unless the user is super-user, which causes *cpio* to retain the owner and group of the files of the previous **cpio –o**. NOTE: If **cpio –i** tries to create a file that already exists and the existing file is the same age or newer, *cpio* will output a warning message and not replace the file. (The **–u** option can be used to unconditionally overwrite the existing file.)

cpio –p (pass) reads the standard input to obtain a list of path names of files that are conditionally created and copied into the destination *directory* tree based upon the options described below.

The meanings of the available options are

–a Reset *access* times of input files after they have been copied. Access times are not reset for linked files when **cpio –pla** is specified.

–b Reverse the order of the *bytes* within each word. Use only with the **–i** option.

−B Input/output is to be blocked 5,120 bytes to the record. The default buffer size is 512 bytes when this and the C options are not used. (**−B** does not apply to the *pass* option; **−B** is meaningful only with data directed to or from a character special device, e.g. **/dev/rmt/0m**.)

−c Write header information in ASCII *character* form for portability. Always use this option when origin and destination machines are different types.

−C *bufsize*

Input/output is to be blocked *bufsize* bytes to the record, where *bufsize* is replaced by a positive integer. The default buffer size is 512 bytes when this and **B** options are not used. (−C does not apply to the *pass* option; −C is meaningful only with data directed to or from a character special device, e.g. **/dev/rmt/0m**.)

−d *directories* are to be created as needed.

−f Copy in all *files* except those in *patterns*. (See the paragraph on **cpio −i** for a description of *patterns*.)

−H Don't follow symbolic links. This option is meaningful only with the **o** or **p** options. Symbolic links, if any, are copied out as such. This is the default.

−I *file* Read the contents of *file* as input. If *file* is a character special device, when the first medium is full replace the medium and type a carriage return to continue to the next medium.

The specified *file* can also reference a remote tape device. A remote tape device name has the form:

*[user@]*system:/dev/???

Where *system* is the remote system, */dev/???* is the particular drive on the remote system (raw, rewinding, non-rewinding, etc.), and the optional *user* is the login name to be used on the remote system, if different from the current user's login name.

Use only with the **-i** option.

−k Attempt to skip corrupted file headers and I/O errors that may be encountered. If you want to copy files from a medium that is corrupted or out of sequence, this option lets you read only those files with good headers. (For *cpio* archives that contain other *cpio* archives, if an error is encountered *cpio* may terminate prematurely. *cpio* will find the next good header, which may be one for a smaller archive, and terminate when the smaller archive's trailer is encountered.) Used only with the **-i** option.

−l Whenever possible, *link* files rather than copying them. Usable only with the **−p** option.

−L Follow symbolic links. This option is meaningful only with the **o** or **p** options. If symbolic links are encountered, the referenced files, rather than the symbolic links themselves, are copied out.

−m Retain previous file *modification* time. This option is ineffective on directories that are being copied.

−M *message*
Define a message to use when switching media. When you use the **−O** or **−I** options and specify a character special device, you can use this option to define the message that is printed when you reach the end of the medium. One **%d** can be placed in the message to print the sequence number of the next medium needed to continue.

−O *file* Direct the output of cpio to *file*. If *file* is a character special device, when the first medium is full replace the medium and type a carriage return to continue to the next medium.

The specified *file* can also reference a remote tape device. A remote tape device name has the form:

*[user@]*system:/dev/???

Where *system* is the remote system, */dev/???* is the particular drive on the remote system (raw, rewinding, non-rewinding, etc.), and the optional *user* is the login name to be used on the remote system, if different from the current user's login name.

Use only with the **−o** option.

−r Interactively *rename* files. If the user types a null line, the file is skipped. If the user types a "." the original pathname will be copied. (Not available with **cpio −p**.)

−s *swap* bytes within each half word. Use only with the **−i** option.

−S *Swap* halfwords within each word. Use only with the **−i** option.

−t Print a *table of contents* of the input. No files are created.

−u Copy *unconditionally* (normally, an older file will not replace a newer file with the same name).

−v *verbose*: causes a list of file names to be printed. When used with the **−t** option, the table of contents looks like the output of an **ls −l** command (see **ls**(1)).

−V *Special*Verbose: print a dot for each file seen. Useful to assure the user that **cpio** is working without printing out all file names.

–6 Process an old (i.e. UNIX System *Sixth* Edition format) file. Use only with the **–i** option.

NOTE: *cpio* assumes four-byte words.

If *cpio* reaches end of medium (end of a diskette for example), when writing to (**–o**) or reading from (**–i**) a character special device, and **–O** and **–I** aren't used, *cpio* will print the message:

If you want to go on, type device/file name when ready.

To continue, you must replace the medium and type the character special device name (**/dev/rdiskette** for example) and carriage return. You may want to continue by directing *cpio* to use a different device. For example, if you have two floppy drives you may want to switch between them so *cpio* can proceed while you are changing the floppies. (A carriage return alone causes the *cpio* process to exit.)

EXAMPLES

The following examples show three uses of *cpio*.

When standard input is directed through a pipe to **cpio –o**, it groups the files so they can be directed (>) to a single file (**../newfile**). The **c** option insures that the file will be portable to other machines. Instead of *ls*(1), you could use *find*(1), *echo*(1), *cat*(1), etc. to pipe a list of names to *cpio*. You could direct the output to a device instead of a file.

```
ls | cpio –oc >../newfile
```

cpio –i uses the output file of **cpio –o** (directed through a pipe with **cat** in the example), extracts those files that match the patterns (**memo/a1**, **memo/b***), creates directories below the current directory as needed (**–d** option), and places the files in the appropriate directories. The **c** option is used when the file is created with a portable header. If no patterns were given, all files from *newfile* would be placed in the directory.

```
cat newfile | cpio –icd "memo/a1" "memo/b*"
```

cpio –p takes the file names piped to it and copies or links (**–l** option) those files to another directory on your machine (*newdir* in the example). The **–d** options says to create directories as needed. The **–m** option says retain the modification time. (It is important to use the **–depth** option of *find*(1) to generate path names for *cpio*. This eliminates problems *cpio* could have trying to create files under read-only directories.)

```
find . –depth –print | cpio –pdlmv newdir
```

SEE ALSO

ar(1), cat(1), echo(1), find(1), ls(1), tar(1).
cpio(4) in the *System Administrator's Reference Manual.*
rmt(1M), rmtops(3)

NOTES

1) Path names are restricted to 256 characters.
2) Only the super-user can copy special files.
3) Blocks are reported in 512-byte quantities.
4) If a file has 000 permissions, contains more than 0 characters of data, and the user is not root, the file will not be saved or restored.

NAME

cpioArchive – an interactive *transferdevice* for performing cpio within the WorkSpace.

SYNOPSIS

cpioArchive menu
cpioArchive versionOK
cpioArchive import
cpioArchive export
cpioArchive dir

DESCRIPTION

The *cpioArchive* command is an interactive tool used for copying files to or from the WorkSpace and a tape drive using cpio. It is a type of *transferdevice,* and therefore understands the generic arguments **menu** and **versionOK.**

The argument **import** is used to copy files from a *cpio* format tape to the WorkSpace's currently selected directory. **xport** is used to copy the files currently selected in the WorkSpace to a tape. When given the argument **dir**, *cpioArchive* will produce a listing of the files on the tape. The *cpioArchive* examines the environment variable $SELECTED, which is set to be the current selection by the WorkSpace. *cpioArchive* can be used to read or write tapes on a remote machine.

FILES

/etc/transferDevice/

SEE ALSO

transfermanager(1G), transferdevice(4), rcpDevice(1), tarArchive(1), workspace(1G), cpio(1)
Programming the IRIS WorkSpace

NAME

cpp – the C language preprocessor

SYNOPSIS

LIBDIR/cpp [option ...] [ifile [ofile]]

DESCRIPTION

The C language preprocessor, *cpp*, is invoked as the first pass of any C compilation by the *cc*(1) command. Thus *cpp's* output is designed to be in a form acceptable as input to the next pass of the C compiler. As the C language evolves, *cpp* and the rest of the C compilation package will be modified to follow these changes. Therefore, the use of *cpp* other than through the *cc*(1) command is not suggested, since the functionality of *cpp* may someday be moved elsewhere. See *m4*(1) for a general macro processor.

cpp optionally accepts two file names as arguments. *Ifile* and *ofile* are respectively the input and output for the preprocessor. They default to standard input and standard output if not supplied.

The following *options* to *cpp* are recognized:

—P
Preprocess the input without producing the line control information used by the next pass of the C compiler.

—C
By default, *cpp* strips C-style comments. If the —C option is specified, all comments (except those found on *cpp* directive lines) are passed along.

—M
Run only the macro preprocessor on the named C programs, requesting it to generate Makefile dependencies and send the result to the standard output.

—U*name*
Remove any initial definition of *name*, where *name* is a reserved symbol that is predefined by the particular preprocessor. Following is the current list of these possibly reserved symbols. The symbols *sgi* and *unix* are always predefined by *cpp*. The symbol *__EXTENSIONS__* is predefined to indicate this is not an ANSI cpp and to make extensions to (proposed) ANSI C visible. The symbols *_SVR3* and *SVR3* which indicate the current version of UNIX are also predefined, as is the symbol *mips*. (The last two predefinitions may change at a later date.) The compiler drivers, *as(1)*, *cc*(1), *pc*(1), and *f77*(1) define *MIPSEB*, *_MIPSEB*, *host_mips*, and the appropriate *LANGUAGE* and *_LANGUAGE* symbols. See the respective man pages for the specific complete list.

The following table shows some names known to be reserved by cpp in at least one machine/environment:

operating system:	unix, dmert, gcos, ibm, os, tss SVR2, SVR3
target hardware:	mips, interdata, pdp11, u370, u3b, u3b5, u3b2, u3b20d, vax
host hardware:	host_mips
languages:	LANGUAGE_C, LANGUAGE_ASSEMBLY, LANGUAGE_PASCAL, LANGUAGE_FORTRAN
UNIX system variant:	RES, RT
lint(1):	lint

—D*name*

—D*name=def*
Define *name* with value *def* as if by a **#define**. If no =*def* is given, *name* is defined with value 1. The **—D** option has lower precedence than the **—U** option. That is, if the same name is used in both a **—U** options and a **—D** option, the name will be undefined regardless of the order of the options.

—I*dir* Change the algorithm for searching for **#include** files whose names do no begin with / to look in *dir* before looking in the directories on the standard list. Thus **#include** files whose names are enclosed in "" will be searched for first in the directory of the file with the **#include** line, then in the directories named in **—I** options, and last in directories on a standard list. For **#include** files whose names are enclosed in <>, the directory of the file with the **#include** line is not searched. If **—I** is given with no *dir*, *cpp* is instructed to suppress the search of the standard list of include directories. This standard list consists only of */usr/include*.

—l When generating **'#lineno filename'** directives for the compiler, use the filename in the first **'#line'** input statement, rather than the name of the input file.

Two special names are understood by *cpp*. The name __LINE__ is defined as the current line number (as a decimal integer) as known by *cpp*, and __FILE__ is defined as the current file name (as a C string) as known by *cpp*. They can be used anywhere (including in macros) just as any other defined name.

All *cpp* directive lines start with # in column 1. Any number of blanks and tabs are allowed between the # and the directive. The directives are:

#define *name token-string*
Replace subsequent instances of *name* with *token-string*.

#define *name(arg, ..., arg) token-string*
Notice that there can be no space between *name* and the (. Replace subsequent instances of *name* followed by a (, a list of comma-separated sets of tokens, and a) followed by *token-string*, where each occurrence of an *arg* in the *token-string* is replaced by the corresponding set of tokens in the comma-separated list. When a macro with arguments is expanded, the arguments are placed into the expanded *token-string* unchanged. After the entire *token-string* has been expanded, *cpp* re-starts its scan for names to expand at the beginning of the newly created *token-string*.

#undef *name*
Cause the definition of *name* (if any) to be forgotten from now on. No additional tokens are permitted on the directive line after *name*.

#ident *"string"*
The *string* and the directive are silently swallowed. No output is produced for this directive.

#pragma
The directive and whatever follows it on the line is passed to the output in a slightly modified form which is not documented. The form may change in a future release.

#pragma once
If this directive appears in an included file, the file will never be included again, even if there is another **#include** of this file. No tokens or comments are permitted after the ‘‘once’’ keyword. Using **#pragma once** is more efficient than using macro wrappers, because the included file is not rescanned, but it may not be portable to third-party preprocessors.

#include *"filename"*

#include *<filename>*
Include at this point the contents of *filename* (which will then be run through *cpp*). When the *<filename>* notation is used, *filename* is only searched for in the standard places. See the **—I** option above for more detail. No additional tokens are permitted on the directive line after the final " or >.

#line *integer-constant filename*

Causes *cpp* to generate line control information for the next pass of the C compiler. *Integer-constant* is the line number of the next line and *filename* is the file from which it comes. If *"filename"* is not given, the current file name is unchanged. No additional tokens are premitted on the directive line after the optional *filename*.

#endif Ends a section of lines begun by a test directive (**#if, #ifdef**, or **#ifndef**). Each test directive must have a matching **#endif**. No additional tokens are permitted on the directive line.

#ifdef *name*

The lines following will appear in the output if and only if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**. No additional tokens are permitted on the directive line after *name*.

#ifndef *name*

The lines following will appear in the output if and only if *name* has not been the subject of a previous **#define**. No additional tokens are permitted on the directive line after *name*.

#if *constant-expression*

Lines following will appear in the output if and only if the *constant-expression* evaluates to non-zero. All binary non-assignment C operators, the ?: operator, the unary —, !, and ~ operators are all legal in *constant-expression*. The precedence of the operators is the same as defined by the C language. There is also a unary operator **defined**, which can be used in *constant-expression* in these two forms: **defined** (*name*) or **defined** *name*. This allows the utility of **#ifdef** and **#ifndef** in a **#if** directive. Only these operators, integer constants, and names which are known by *cpp* should be used in *constant-expression*. In particular, the **sizeof** operator is not available.

To test whether either of two symbols, *foo* and *fum*, are defined, use:

```
#if defined(foo) || defined(fum)
```

#else The lines following will appear in the output if and only if the preceding test directive evaluates to zero. No additional tokens are permitted on the directive line.

The test directives and the possible **#else** directives can be nested.

FILES

INCDIR	standard directory list for **#include** files, usually /usr/include
LIBDIR	usually /usr/lib

SEE ALSO

cc(1), line(1), m4(1)

NAME

cps – construct C to PostScript interface

SYNOPSIS

cps [**–c**] [**–D** *symbol*] [**–I** *filename*] [**–i**] [*PostScript file*]

DESCRIPTION

NeWS compiles a specification file containing C procedure names and PostScript code into a header file *filename.h* that can be included in C programs. Only one input file can be specified, and if the *filename.h* file has been previously created, a backup copy of this file will be generated in the form *filename.h.BAK* before the new file is generated.

The convention is for the input specification file to end in *.cps*.

OPTIONS

–c Compiles a PostScript file for faster loading by NeWS, and is not used to generate a specification file for programs. For example, the following command line:

```
cps -c < input_file > output_file
```

will convert the *input_file* from the ascii form of the PostScript Language, to the compressed binary form. When read by NeWS, the *output_file* will execute exactly the same as *input_file*, except that it will be faster. The **–c** option **will not** work if the *input_file* uses constructs like currentfile readstring, which are often used with the image primitive.

–D *symbol* Defines symbols to be passed onto the C language pre-processor (*cpp*) which processes the input file.

–I *filename* Specifies include files. Passed on to the pre-processor.

–i Generates two specification files: one that contains only the C procedures and PostScript code that are user-defined, and one that contains other definitions required for the C-PostScript interface. For example, **ps_open_PostScript** and **ps_close_PostScript** would be defined in the second file. The second file references the user-defined procedures as **extern char**. The first file is of the form *filename.c*, and the second file is of the form *filename.h*. already exist.

This option is valuable for controlling the size of the CPS include files in multiple source files. The *filename.h* would only need to be included once. Each source file would only need to include its specific *filename.c* file generated by this option.

SEE ALSO

cpp(1).
4Sight User's Guide, Section 2, "Programming in NeWS.

TRADEMARK

PostScript is a registered trademark of Adobe Systems, Inc.

NAME

crontab – user crontab file

SYNOPSIS

crontab [file]
crontab –r
crontab –l

DESCRIPTION

crontab copies the specified file, or standard input if no file is specified, into a directory that holds all users' crontabs. The –r option removes a user's crontab from the crontab directory. *crontab* –l will list the crontab file for the invoking user.

Users are permitted to use *crontab* if their names appear in the file **/usr/lib/cron/cron.allow.** If that file does not exist, the file **/usr/lib/cron/cron.deny** is checked to determine if the user should be denied access to *crontab.* If neither file exists, only root is allowed to submit a job. If **cron.allow** does not exist and **cron.deny** exists but is empty, global usage is permitted. The allow/deny files consist of one user name per line.

A crontab file consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns that specify the following:

> minute (0–59),
> hour (0–23),
> day of the month (1–31),
> month of the year (1–12),
> day of the week (0–6 with 0=Sunday).

Each of these patterns may be either an asterisk (meaning all legal values) or a list of elements separated by commas. An element is either a number or two numbers separated by a minus sign (meaning an inclusive range). Note that the specification of days may be made by two fields (day of the month and day of the week). If both are specified as a list of elements, both are adhered to. For example, 0 0 1,15 * 1 would run a command on the first and fifteenth of each month, as well as on every Monday. To specify days by only one field, the other field should be set to * (for example, 0 0 * * 1 would run a command only on Mondays).

The sixth field of a line in a crontab file is a string that is executed by the shell at the specified times. A percent character in this field (unless escaped by \) is translated to a new-line character. Only the first line (up to a % or end of line) of the command field is executed by the shell. The other lines are made available to the command as standard input.

The shell is invoked from your $HOME directory with an **arg0** of **sh.** Users who desire to have their *.profile* executed must explicitly do so in the crontab file. *Cron* supplies a default environment for every shell, defining **HOME, LOGNAME, USER, SHELL(=/bin/sh), PATH(=/usr/sbin:/usr/bsd/:/usr/bin:/bin/:/etc:/usr/etc),** and **TZ.**

If you do not redirect the standard output and standard error of your commands, any generated output or errors will be mailed to you.

Any errors encountered while parsing the crontab file (or stdin) will cause *crontab* to abort with no changes being made to any existing crontab.

FILES

/usr/lib/cron	main cron directory
/usr/spool/cron/crontabs	spool area
/usr/lib/cron/log	accounting information
/usr/lib/cron/cron.allow	list of allowed users
/usr/lib/cron/cron.deny	list of denied users

SEE ALSO

sh(1).
cron(1M) in the *System Administrator's Reference Manual.*

WARNINGS

If you inadvertently enter the **crontab** command with no argument(s), and do not wish to overwrite your crontab, you may get out by either exiting with DEL or hitting ^D.

If a job is scheduled during the 'witching hour' - the time during a change from the main to alternate time zone, the job will either be run once (if the actual time exists twice) or not at all (if the actual time never exists).

NAME

crypt – encode/decode

SYNOPSIS

crypt [password]
crypt [–k]

DESCRIPTION

crypt reads from the standard input and writes on the standard output. The *password* is a key that selects a particular transformation. If no argument is given, *crypt* demands a key from the terminal and turns off printing while the key is being typed in. If the –k option is used, *crypt* will use the key assigned to the environment variable CRYPTKEY. *crypt* encrypts and decrypts with the same key:

```
crypt key <clear >cypher
crypt key <cypher | pr
```

Files encrypted by *crypt* are compatible with those treated by the editors *ed*(1), *edit*(1), *ex*(1), and *vi*(1) in encryption mode.

The security of encrypted files depends on three factors: the fundamental method must be hard to solve; direct search of the key space must be infeasible; "sneak paths" by which keys or clear text can become visible must be minimized.

crypt implements a one-rotor machine designed along the lines of the German Enigma, but with a 256-element rotor. Methods of attack on such machines are known, but not widely; moreover the amount of work required is likely to be large.

The transformation of a key into the internal settings of the machine is deliberately designed to be expensive, i.e., to take a substantial fraction of a second to compute. However, if keys are restricted to (say) three lowercase letters, then encrypted files can be read by expending only a substantial fraction of five minutes of machine time.

If the key is an argument to the *crypt* command, it is potentially visible to users executing *ps*(1) or a derivative. The choice of keys and key security are the most vulnerable aspect of *crypt*.

FILES

/dev/tty	for typed key

SEE ALSO

ed(1), edit(1), ex(1), makekey(1), ps(1), stty(1), vi(1).

WARNING

This command is provided with the Security Administration Utilities, which is only available in the United States. If two or more files encrypted with the same key are concatenated and an attempt is made to decrypt the result, only the contents of the first of the original files will be decrypted correctly.

BUGS

If output is piped to *nroff* and the encryption key is *not* given on the command line, *crypt* can leave terminal modes in a strange state (see *stty*(1)).

NAME

csh – a shell (command interpreter) with C-like syntax

SYNOPSIS

csh [**–bcefinstvVxX**] [arg ...]

DESCRIPTION

csh is a command language interpreter incorporating a history mechanism (see **History Substitutions**), job control facilities (see **Jobs**), interactive file name and user name completion (see **File Name Completion**), and a C-like syntax.

Login shells begin by executing commands from the file */etc/cshrc*. Every instance of *csh* will then execute the commands in the file *.cshrc* in the home directory of the invoker. If it is a login shell, it then also executes commands from the file *.login* there. It is typical for users on CRTs to invoke *tset*(1) from their *.login* file.

In the normal case, the shell will then begin reading commands from the terminal, prompting with %. Processing of arguments and the use of the shell to process files containing command scripts will be described later.

The shell then repeatedly performs the following actions: a line of command input is read and broken into words. This sequence of words is placed on the command history list and then parsed. Finally each command in the current line is executed.

When a login shell terminates, it executes commands from the file *.logout* in the user's home directory.

Lexical Structure

The shell splits input lines into words at blanks and tabs with the following exceptions. The characters &, |, ;, <, >, (,), form separate words. If doubled in &&, | |, << or >>, these pairs form single words. These parser metacharacters may be made part of other words, or their special meaning may be prevented, by preceding them with a backslash (\). A newline preceded by a \ is equivalent to a blank. It is usually necessary to use the backslash to escape the parser metacharacters when you want to use them literally rather than as metacharacters.

Strings enclosed in matched pairs of quotation marks, either single or double quotation marks (', ‘ or ") form parts of a word. Metacharacters in these strings, including blanks and tabs, do not form separate words. Such quotations have semantics to be described subsequently.

Within pairs of single or double quotation marks, a newline (carriage return) preceded by a \ gives a true newline character. This is used to set up a file of strings separated by newlines, as for *fgrep*.

When the shell's input is not a terminal, the character # introduces a comment which continues to the end of the input line. It is prevented from having this special meaning when preceded by \ or if bracketed by a pair of single or double quotation marks.

Commands

A simple command is a sequence of words, the first of which specifies the command to be executed.

A simple command or a sequence of simple commands separated by | characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next.

Sequences of pipelines may be separated by ;, and are then executed sequentially. A sequence of pipelines may be executed without immediately waiting for it to terminate by following it with an &, which means to run it in background.

Parentheses (and) around a pipeline or sequence of pipelines cause the whole series to be treated as a simple command, which may in turn be a component of a pipeline, etc. It is also possible to separate pipelines with || or && indicating, as in the C language, that the second is to be executed only if the first fails or succeeds, respectively. (See **Expressions.**)

Jobs

The shell associates a *job* with each pipeline. It keeps a table of current jobs, printed by the *jobs* command, and assigns them small integer numbers. When a job is started asynchronously with '&', the shell prints a line which looks like:

```
[1] 1234
```

indicating that the job which was started asynchronously was job number 1 and had one (top-level) process, whose process id was 1234.

If you are running a job and wish to do something else you may hit the key ^Z (control-Z) which sends a STOP signal to the current job. The shell will then normally indicate that the job has been 'Stopped', and print another prompt. You can then manipulate the state of this job, putting it in the background with the *bg* command, or run some other commands and then eventually bring the job back into the foreground with the foreground command *fg*. A ^Z takes effect immediately and is like an interrupt in that pending output and unread input are discarded when it is typed.

A job being run in the background will stop if it tries to read from the terminal. Background jobs are normally allowed to produce output, but this can be disabled by giving the command "stty tostop" (see *stty*(1)). If you set this tty option, then background jobs will stop when they try to produce output, just as they do when they try to read input. Note that in the default mode (``stty -tostop''), background jobs are only allowed to write to the terminal without blocking. Background jobs that attempt to change the operational mode of their controlling terminal using *ioctl*(2) system calls will be stopped, regardless of the setting of ``tostop'' mode. *Ioctl* calls which only return information without altering the state of the terminal will not cause a background process to be stopped (refer to *termio*(7) for more information).

There are several ways to refer to jobs in the shell. The character '%' introduces a job name. If you wish to refer to job number 1, you can name it as '%1'. Just naming a job brings it to the foreground; thus '%1' is a synonym for 'fg %1', bringing job 1 back into the foreground. Similarly saying '%1 &' resumes job 1 in the background. Jobs can also be named by prefixes of the string typed in to start them, if these prefixes are unambiguous. For example, '%ex' would normally restart a suspended *ex*(1) job, if there were only one suspended job whose name began with the string 'ex'. It is also possible to say '%?string' which specifies a job whose text contains *string*, if there is only one such job.

The shell maintains a notion of the current and previous jobs. In output pertaining to jobs, the current job is marked with a '+' and the previous job with a '–'. The abbreviation '%+' refers to the current job and '%–' refers to the previous job. For close analogy with the syntax of the *history* mechanism (described below), '%%' is also a synonym for the current job.

Status Reporting

This shell learns immediately whenever a process changes state. It normally informs you whenever a job becomes blocked so that no further progress is possible, but only just before it prints a prompt. This is done so that it does not otherwise disturb your work. If, however, you set the shell variable *notify*, the shell will notify you immediately of changes of status in background jobs. There is also a shell command *notify* which marks a single process so that its status changes will be immediately reported. By default *notify* marks the current process (the one that is referred to by '%'). You may also apply *notify* to a particular job (e.g., ``notify %2'').

When you try to leave the shell while jobs are stopped, you will be warned that 'You have stopped jobs.' You may use the *jobs* command to see what they are. If you do this or immediately try to exit again, the shell will not warn you a second time, and the suspended jobs will be terminated.

File Name Completion

When the file name completion feature is enabled by setting the shell variable *filec* (see set), *csh* will interactively complete file names and user names from unique prefixes, when they are input from the terminal followed by the escape character (the escape key, or control-[). For example, if the current directory looks like

```
DSC.OLD    bin         cmd         lib         xmpl.c
```

```
    DSC.NEW   chaosnet   cmtest   mail   xmpl.o
    bench     class      dev      mbox   xmpl.out
```
and the input is
```
    % vi ch<escape>
```
csh will complete the prefix "ch" to the only matching file name "chaosnet", changing the input line to
```
    % vi chaosnet
```
However, given
```
    % vi D<escape>
```
csh will only expand the input to
```
    % vi DSC.
```
and will sound the terminal bell to indicate that the expansion is incomplete, since there are two file names matching the prefix "D".

If a partial file name is followed by ^D (control-D), then, instead of completing the name, *csh* will list all file names matching the prefix. For example, the input
```
    % vi D<control-D>
```
causes all files beginning with "D" to be listed:
```
    DSC.NEW   DSC.OLD
```
while the input line remains unchanged.

The same system of escape and control-D can also be used to expand partial user names, if the word to be completed (or listed) begins with the character "~". For example, typing
```
    cd ~ro<control-D>
```
may produce the expansion
```
    cd ~root
```

The use of the terminal bell to signal errors or multiple matches can be inhibited by setting the variable *nobeep*.

Normally, all files in the particular directory are candidates for name completion. Files with certain suffixes can be excluded from consideration by setting the variable *fignore* to the list of suffixes to be ignored. Thus, if *fignore* is set by the command
```
    % set fignore = (.o .out)
```
then typing
```
    % vi x<escape>
```
would result in the completion to
```
    % vi xmpl.c
```
ignoring the files "xmpl.o" and "xmpl.out". However, if the only completion possible requires not ignoring these suffixes, then they are not ignored. In addition, *fignore* does not affect the listing of file names by control-D. All files are listed regardless of their suffixes.

Note that it is not possible to substitute different characters for the escape character and control-D in file completion. Activating file completion by setting *filec* also causes the end-of-line character for the terminal (the EOL character as described in *termio*(7)) to become undefined.

Substitutions

We now describe the various transformations the shell performs on the input in the order in which they occur.

History substitutions

History substitutions place words from previous command input as portions of new commands, making it easy to repeat commands, repeat arguments of a previous command in the current command, or fix spelling mistakes in the previous command with little typing and a high degree of confidence.

History substitutions begin with the character ! and may begin anywhere in the input stream (with the proviso that they may not nest.)

This ! may be preceded by a \ to turn off its special meaning; for convenience, a ! is also passed unchanged when it is followed by a blank, tab, newline, = or (.

Therefore, do not put a space after the ! and the command reference when you are invoking the shell's history mechanism. (History substitutions also occur when an input line begins with ^. This special abbreviation will be described later.)

An input line which invokes history substitution is echoed on the terminal before it is executed, as it would look if typed out in full.

The shell's history list, which may be seen by typing the *history* command, contains all commands input from the terminal which consist of one or more words. History substitutions reintroduce sequences of words from these saved commands into the input stream. The *history* variable controls the size of the input stream. The previous command is always retained, regardless of its value. Commands are numbered sequentially from 1.

Consider the following output from the *history* command:

```
 9  write michael
10  ex write.c
11  cat oldwrite.c
12  diff *write.c
```

The commands are shown with their event numbers. It is not usually necessary to use event numbers, but the current event number can be made part of the *prompt* by placing an \! in the prompt string. This is done by setting prompt = \! and the prompt character of your choice.

For example, if the current event is number 13, we can call up the command recorded as event 11 in several ways: **!–2** [i.e., 13–2]; by the first letter of one of its command words, such as **!c** referring to the c in *cat*; or **!wri** for event 9, or by a string contained in a word in the command as in **!?mic?** also referring to event 9.

These forms, without further modification, simply reintroduce the words of the specified events, each separated by a single blank. As a special case **!!** refers to the previous command; thus **!!** alone is essentially a *redo*.

Words are selected from a command event and acted upon according to the following formula:

event:position:action

The *event* is the command you wish to retrieve. As mentioned above, it may be summoned up by event number and in several other ways. All that the *event* notation does is to tell the shell which command you have in mind.

Position picks out the words from the command event on which you want the *action* to take place. The *position* notation can do anything from altering the command completely to making some very minor substitution, depending on which words from the command event you specify with the *position* notation.

To select words from a command event, follow the event specification with a : and a designator (by position) for the desired words.

The words of a command event are picked out by their position in the input line. Positions are numbered from 0, the first word (usually command) being position 0, the second word having position 1, and so forth. If you designate a word from the command event by stating its position, it means you want to include it in your revised command. All the words that you want to include in a revised command must be designated by position notation in order to be included.

The basic position designators are:

0	first (command) word
n	*n*th argument
^	first argument, i.e., 1
$	last argument
%	matches the word of an ?s? search which immediately precedes it; used to strip one word out of a command event for use in another command. Example: **!?four?:%:p** prints **four**.

x–*y*	range of words (e.g., 1–3 means from position 1 to position 3).
–*y*	abbreviates 0–*y*
*	stands for ^–$, or indicates position 1 if only one word in event.
*x**	abbreviates *x*–$ where *x* is a position number.
x–	like *x** but omitting last word $

The : separating the event specification from the word designator can be omitted if the argument selector begins with a ^, $, *, – or %.

Modifiers, each preceded by a :, may be used to act on the designated words in the specified command event. The following modifiers are defined:

h	Remove a trailing pathname component, leaving the head.
r	Remove a trailing .xxx component, leaving the root name.
e	Remove all but the extension .xxx part.
s/*old*/*new*/	Substitute *new* for *old*
t	Remove all leading pathname components, leaving the tail.
&	Repeat the previous substitution.
g	Apply the change globally, prefixing the above, e.g., g&.
p	Print the new command but do not execute it.
q	Quote the substituted words, preventing further substitutions.
x	Like q, but break into words at blanks, tabs and newlines.

Unless preceded by a g, the modification is applied only to the first modifiable word. With substitutions it is an error for no word to be applicable.

The left hand side of substitutions are not regular expressions in the sense of the editors, but rather strings. Any character may be used as the delimiter in place of /; a \ quotes the delimiter into the *l* and *r* strings. The character & in the right hand side is replaced by the text from the left. A \ quotes & also. A null *l* uses the previous string either from a *l* or from a contextual scan string *s* in !?*s*?. The trailing delimiter in the substitution may be omitted if (but only if) a newline follows immediately as may the trailing ? in a contextual scan.

A history reference may be given without an event specification, e.g., !$. In this case the reference is to the previous command. If a previous history reference occurred on the same line, this form repeats the previous reference. Thus !?foo?^ !$ gives the first and last arguments from the command matching ?foo?.

You can quickly make substitutions to the previous command line by using the ^ character as the first non-blank character of an input line. This is equivalent to **!:s^** providing a convenient shorthand for substitutions on the text of the previous line. Thus **^lb^lib** fixes the spelling of lib in the previous command. Finally, a history substitution may be surrounded with { and } if necessary to insulate it from the characters which follow. Thus, after **ls –ld ~paul** we might do **!{l}a** to do **ls –ld ~paula**, while **!la** would look for a command starting **la**.

Quotations with ' and "

The quotation of strings by ' and " can be used to prevent all or some of the remaining substitutions which would otherwise take place if these characters were interpreted as metacharacters or wild card matching characters. Strings enclosed in single quotes (') are prevented any further interpretation or expansion. Strings enclosed in " may still be variable and command expanded as described below.

In both cases the resulting text becomes (all or part of) a single word; only in one special case (see Command Substitution below) does a " quoted string yield parts of more than one word; ' quoted strings never do.

Alias substitution

The shell maintains a list of aliases which can be established, displayed and modified by the *alias* and *unalias* commands. After a command line is scanned, it is parsed into distinct commands and the first word of each command, left-to-right, is checked to see if it has an alias. If it does, then the text which is the alias for that command is reread with the history mechanism available as though that command were the previous input line. The resulting words replace the command and argument list. If no reference is made to the history list, then the argument list is left unchanged.

Thus if the alias for *ls* is **ls –l** the command **ls /usr** would map to **ls -l /usr** , the argument list here being undisturbed. Similarly if the alias for *lookup* was **grep !^ /etc/passwd**, then **lookup bill** would map to **grep bill /etc/passwd**.

If an alias is found, the word transformation of the input text is performed and the aliasing process begins again on the reformed input line. Looping is prevented if the first word of the new text is the same as the old by flagging it to prevent further aliasing. Other loops are detected and cause an error.

Note that the mechanism allows aliases to introduce parser metasyntax. Thus we can **alias print ´pr \!* | lpr´** to make a command which *pr* s its arguments to the line printer.

Variable substitution

The shell maintains a set of variables, each of which has as value a list of zero or more words. Some of these variables are set by the shell or referred to by it. For instance, the *argv* variable is an image of the shell's argument list, and words of this variable's value are referred to in special ways.

The values of variables may be displayed and changed by using the *set* and *unset* commands. Of the variables referred to by the shell a number are toggles; the shell does not care what their value is, only whether they are set or not. For instance, the *verbose* variable is a toggle which causes command input to be echoed. The setting of this variable results from the –v command line option.

Other operations treat variables numerically. The @ command permits numeric calculations to be performed and the result assigned to a variable. Variable values are, however, always represented as (zero or more) strings. For the purposes of numeric operations, the null string is considered to be zero, and the second and subsequent words of multiword values are ignored.

After the input line is aliased and parsed, and before each command is executed, variable substitution is performed keyed by $ characters. This expansion can be prevented by preceding the $ with a \ except within double quotes (") where it always occurs, and within single quotes (') where it never occurs. Strings quoted by ` are interpreted later (see Command substitution below) so $ substitution does not occur there until later, if at all. A $ is passed unchanged if followed by a blank, tab, or end-of-line.

Input/output redirections are recognized before variable expansion, and are variable expanded separately. Otherwise, the command name and entire argument list are expanded together. It is thus possible for the first (command) word to this point to generate more than one word, the first of which becomes the command name, and the rest of which become arguments.

Unless enclosed in double quotes or given the *:q* modifier, the results of variable substitution may eventually be command and filename substituted. Within double quotes, a variable whose value consists of multiple words expands to a (portion of) a single word, with the words of the variables value separated by blanks. When the *:q* modifier is applied to a substitution, the variable will expand to multiple words with each word separated by a blank and quoted to prevent later command or filename substitution.

Metasequences for variable substitution

The following metasequences are provided for introducing variable values into the shell input. Except as noted, it is an error to reference a variable which is not set.

$name
${name}

Are replaced by the words of the value of variable *name*, each separated by a blank. Braces insulate *name* from following characters which would otherwise be part of it. Shell variables have names consisting of up to 20 letters and digits starting with a letter. The underscore character is considered a letter.

If *name* is not a shell variable, but is set in the environment, then that value is returned (but : modifiers and the other forms given below are not available in this case).

$name[selector]
${name[selector]}

May be used to select only some of the words from the value of *name*. The selector is subjected to $ substitution and may consist of a single number or two numbers separated by a –. The first word of a variables value is numbered 1. If the first number of a range is omitted it defaults to 1. If the last member of a range is omitted it defaults to $#name. The selector * selects all words. It is not an error for a range to be empty if the second argument is omitted or in range.

$#name
${#name}

Gives the number of words in the variable. This is useful for later use in a [*selector*].

$0

Substitutes the name of the file from which command input is being read. An error occurs if the name is not known.

$number
${number}

Equivalent to $argv [number].

$*

Equivalent to $argv [*].

The modifiers :h, :t, :r, :q and :x may be applied to the substitutions above as may :gh, :gt and :gr. If braces { } appear in the command form, then the modifiers must appear within the braces. The current implementation allows only one : modifier on each $ expansion.

The following substitutions may not be modified with : modifiers.

$?name
${?name}
Substitutes the string 1 if name is set, 0 if it is not.

$?0
Substitutes 1 if the current input filename is known, 0 if it is not.

$$
Substitute the (decimal) process number of the (parent) shell.

$<
Substitutes a line from the standard input, with no further interpretation thereafter. It can be used to read from the keyboard in a shell script.

Command and filename substitution

The remaining substitutions, command and filename substitution, are applied selectively to the arguments of built-in commands. This means that portions of expressions which are not evaluated are not subjected to these expansions. For commands which are not internal to the shell, the command name is substituted separately from the argument list. This occurs very late, after input-output redirection is performed, and in a child of the main shell.

Command substitution

Command substitution is indicated by a command enclosed in ‘. The output from such a command is normally broken into separate words at blanks, tabs and newlines, with null words being discarded, this text then replacing the original string. Within double quotes ("), only newlines force new words; blanks and tabs are preserved.

In any case, the single final newline does not force a new word. Note that it is thus possible for a command substitution to yield only part of a word, even if the command outputs a complete line.

Filename substitution

If a word contains any of the characters *, ?, [or { or begins with the character ~, then that word is a candidate for filename substitution, also known as ‘‘globbing’’. This word is then regarded as a pattern, and replaced with an alphabetically sorted list of file names which match the pattern. In a list of words specifying filename substitution it is an error for no pattern to match an existing file name, but it is not required for each pattern to match. Only the metacharacters *, ? and [imply pattern matching, the characters ~ and { being more akin to abbreviations.

In matching filenames, the character . at the beginning of a filename or immediately following a /, as well as the character / must be matched explicitly. The character * matches any string of characters, including the null string. The character ? matches any single character. The sequence [...] matches any one of the characters enclosed. Within [...], a pair of characters separated by – matches any character lexically between the two.

The character ˜ at the beginning of a filename is used to refer to home directories. Standing alone, i.e., ˜ it expands to the invokers home directory as reflected in the value of the variable *home*. When followed by a name consisting of letters, digits and – characters, the shell searches for a user with that name and substitutes their home directory; thus **˜ken** might expand to **/usr/ken** and **˜ken/chmach** to **/usr/ken/chmach**. If the character ˜ is followed by a character other than a letter or / or appears not at the beginning of a word, it is left undisturbed.

The metanotation **a{b,c,d}e** is a shorthand for **abeaceade**. Left to right order is preserved, with results of matches being sorted separately at a low level to preserve this order. This construct may be nested. Thus **˜source/s1/{oldls,ls}.c** expands to **/usr/source/s1/oldls.c /usr/source/s1/ls.c** whether or not these files exist without any chance of error if the home directory for **source** is **/usr/source**. Similarly **../{memo,*box}** might expand to **../memo ../box ../mbox**. (Note that memo was not sorted with the results of matching *box.) As a special case {, } and {} are passed undisturbed.

Input/output

The standard input and standard output of a command may be redirected with the following syntax:

< name

Open file *name* (which is first variable, command and filename expanded) as the standard input.

<< word

Read the shell input up to a line which is identical to *word*. *word* is not subjected to variable, filename or command substitution, and each input line is compared to *word* before any substitutions are done on this input line. Unless a quoting \, ", ´ or ` appears in *word*, variable and command substitution is performed on the intervening lines, allowing \ to quote $, \ and `. Commands which are substituted have all blanks, tabs, and newlines preserved, except for the final newline which is dropped. The resultant text is placed in an anonymous temporary file which is given to the command as standard input.

> name
>! name
>& name
>&! name

The file *name* is used as standard output. If the file does not exist then it is created; if the file exists, it is truncated, its previous contents being lost.

If the variable *noclobber* is set, then the file must not exist or be a character special file (e.g., a terminal or */dev/null*) or an error results. This helps prevent accidental destruction of files. In this case the ! forms can be used and suppress this check.

The forms involving &, route the diagnostic output into the specified file as well as the standard output. *Name* is expanded in the same way as < input filenames are.

>> name
>>& name
>>! name
>>&! name

Uses file *name* as standard output like > but places output at the end of the file. If the variable *noclobber* is set, then it is an error for the file not to exist unless one of the ! forms is given. Otherwise similar to >.

A command receives the environment in which the shell was invoked as modified by the input-output parameters and the presence of the command in a pipeline. Thus, unlike some previous shells, commands run from a file of shell commands have no access to the text of the commands by default; rather they receive the original standard input of the shell. The << mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block read its input.

Diagnostic output may be directed through a pipe with the standard output. Simply use the form |& rather than just |. To redirect standard output and standard error to separate files, use (cmd > file1) >& file2; */dev/tty* may be used to redirect input or output to or from your terminal.

Expressions

A number of the built-in commands (to be described subsequently) take expressions, in which the operators are similar to those of C, with the same precedence. These expressions appear in the @, *exit*, *if*, and *while* commands. The following operators are available:

|| && | ^ & == != =~ !~ <= >= < > << >> + − * / % ! ~ ( )

Here the precedence increases to the right, ==, !=, =~ and !~; <=, >=, < and >; << and >>; + and −; *, / and % being, in groups, at the same level. The ==, !=, =~ and !~ operators compare their arguments as strings; all others operate on numbers. The operators =~ and !~ are like != and == except that the right hand side is a *pattern* (which may contain *, ? and instances of [...]) against which the left hand operand is matched. This reduces the need for use of the *switch* statement in shell scripts when all that is really needed is pattern matching.

Strings which begin with 0 are considered octal numbers. Null or missing arguments are considered 0. The result of all expressions are strings, which represent decimal numbers. It is important to note that no two components of an expression can appear in the same word; except when adjacent to components of expressions which are syntactically significant to the parser (& | < > ( )) they should be surrounded by spaces.

Command executions can be used as primitive operands in expressions. When used in an expression, the command is enclosed in { and }, e.g., {command}. Command executions succeed, returning true (1), if the command exits with status 0, otherwise they fail, returning false (0). If more detailed status information is required, then the command should be executed outside of an expression and the variable *status* examined.

File enquiries can also be used as primitive operands in expressions. They should be of the form −*l name* where *l* is one of:

r	read access
w	write access
x	execute access
e	existence
o	ownership
z	zero size
f	plain file
d	directory
l	symbolic link
c	character special file
b	block special file
p	named pipe (fifo)
u	set-user-ID bit is set
g	set-group-ID bit is set
k	sticky bit is set
s	size greater than zero

t open file descriptor for terminal device

The specified name is command and filename expanded and then tested to see if it has the specified relationship to the real user. If the file does not exist or is inaccessible, then all enquiries return false (0).

Control Flow

The shell contains a number of commands which can be used to regulate the flow of control in command files (shell scripts) and (in limited but useful ways) from terminal input. These commands all operate by forcing the shell to reread or skip in its input and, due to the implementation, restrict the placement of some of the commands.

The *foreach*, *switch*, and *while* statements, as well as the *if–then–else* form of the *if* statement require that the major keywords appear in a single simple command on an input line as shown below.

If the shell's input is not seekable, the shell buffers up input whenever a loop is being read and performs seeks in this internal buffer to accomplish the rereading implied by the loop. (To the extent that this allows, backward *goto* s will succeed on non-seekable inputs.)

Built-in Commands

Built-in commands are executed within the shell. If a built-in command occurs as any component of a pipeline except the last, then it is executed in a subshell.

alias
alias name
alias name wordlist

The first form prints all aliases. The second form prints the alias for name. The final form assigns the specified *wordlist* as the alias of *name*; *wordlist* is command and filename substituted. *name* is not allowed to be *alias* or *unalias*.

alloc

Shows the amount of dynamic memory acquired, broken down into used and free memory. With an argument shows the number of free and used blocks in each size category. The categories start at size 8 and double at each step. This command's output may vary across system types, since systems other than the VAX may use a different memory allocator.

bg
bg %job ...

Puts the current or specified jobs into the background, continuing them if they were stopped.

break
Causes execution to resume after the *end* of the nearest enclosing *foreach* or *while*. The remaining commands on the current line are executed. Multi-level breaks are thus possible by writing them all on one line.

breaksw
Causes a break from a *switch*, resuming after the *endsw*.

case label:
A label in a *switch* statement as discussed below.

cd
cd name
chdir
chdir name
Change the shell's working directory to directory *name*. If no argument is given, then change to the home directory of the user.

If *name* is not found as a subdirectory of the current directory (and does not begin with /, ./ or ../), then each component of the variable *cdpath* is checked to see if it has a subdirectory *name*. Finally, if all else fails but *name* is a shell variable whose value begins with /, then this is tried to see if it is a directory.

continue
Continue execution of the nearest enclosing *while* or *foreach*. The rest of the commands on the current line are executed.

default:
Labels the default case in a *switch* statement. The default should come after all *case* labels.

dirs
dirs –l
Prints the directory stack; the top of the stack is at the left, the first directory in the stack being the current directory. In the first form the user's home directory is represented by ˜.

echo wordlist
echo –n wordlist
The specified words are written to the shell's standard output, separated by spaces, and terminated with a newline unless the **–n** option or the \c escape is specified. The following C-like escape sequences are available:

\b	backspace
\c	print line without new-line
\f	form-feed

\n	new-line
\r	carriage return
\t	tab
\\	backslash
\n	the character whose ASCII code is the 1-, 2- or 3-digit octal number *n*.

else
end
endif
endsw

See the description of the *foreach*, *if*, *switch*, and *while* statements below.

eval arg ...

(As in *sh*(1).) The arguments are read as input to the shell and the resulting command(s) executed in the context of the current shell. This is usually used to execute commands generated as the result of command or variable substitution, since parsing occurs before these substitutions. See *tset*(1) for an example of using *eval*.

exec command

The specified command is executed in place of the current shell.

exit
exit(expr)

The shell exits either with the value of the *status* variable (first form) or with the value of the specified *expr* (second form).

fg
fg %job ...

Brings the current or specified jobs into the foreground, continuing them if they were stopped.

foreach name (wordlist)
 ...
end

The variable *name* is successively set to each member of *wordlist* and the sequence of commands between this command and the matching *end* are executed. (Both *foreach* and *end* must appear alone on separate lines.)

The built-in command *continue* may be used to continue the loop prematurely and the built-in command *break* to terminate it prematurely. When this command is read from the terminal, the loop is read up once prompting with ? before any statements in the loop are executed. If you make a mistake typing in a loop at the terminal, you can rub it out.

glob wordlist

Like *echo* but no \ escapes are recognized and words are delimited by null characters in the output. Useful for programs which wish to use the shell to filename expand a list of words.

goto word

The specified *word* is filename and command expanded to yield a string of the form label. The shell rewinds its input as much as possible and searches for a line of the form label: possibly preceded by blanks or tabs. Execution continues after the specified line.

history
history *n*
history –r *n*
history –h *n*

Displays the history event list; if *n* is given only the *n* most recent events are printed. The **–r** option reverses the order of printout to be most recent first rather than oldest first. The **–h** option causes the history list to be printed without leading numbers. This is used to produce files suitable for sourceing using the –h option to *source*.

if (expr) command

If the specified expression evaluates true, then the single *command* with arguments is executed. Variable substitution on *command* happens early, at the same time it does for the rest of the *if* command. *Command* must be a simple command, not a pipeline, a command list, or a parenthesized command list. Input/output redirection occurs even if *expr* is false, when command is not executed (this is a bug).

if (expr) **then**
...
else if (expr2) **then**
...
else
...
endif

If the specified *expr* is true, then the commands to the first *else* are executed; else if *expr2* is true, then the commands to the second else are executed, etc. Any number of *else-if* pairs are possible; only one *endif* is needed. The *else* part is likewise optional. (The words *else* and *endif* must appear at the beginning of input lines; the *if* must appear alone on its input line or after an *else*.)

jobs

jobs −l

Lists the active jobs; given the **−l** options lists process id's in addition to the normal information.

kill %job
kill −sig %job ...
kill pid
kill −sig pid ...
kill −l

Sends either the TERM (terminate) signal or the specified signal to the specified jobs or processes. Signals are either given by number or by names (as given in */usr/include/signal.h,* stripped of the prefix "SIG"). The signal names are listed by "kill −l". There is no default, saying just 'kill' does not send a signal to the current job. If the signal being sent is TERM (terminate) or HUP (hangup), then the job or process will be sent a CONT (continue) signal as well.

limit
limit *resource*
limit *resource maximum-use*
limit −h
limit −h *resource*
limit −h *resource maximum-use*

Limits the consumption by the current process and each process it creates to not individually exceed *maximum-use* on the specified *resource*. If no *maximum-use* is given, then the current limit is printed; if no *resource* is given, then all limitations are given. If the **−h** flag is given, the hard limits are used instead of the current limits. The hard limits impose a ceiling on the values of the current limits. Only the super-user may raise the hard limits, but a user may lower or raise the current limits within the legal range.

Resources controllable currently include *cputime*, the maximum number of cpu-seconds to be used by each process, *filesize*, the largest single file which can be created, *datasize*, the maximum growth of the data+stack region via *sbrk*(2) beyond the end of the program text, *stacksize*, the maximum size of the automatically-extended stack region, *coredumpsize*, the size of the largest core dump that will be created, and *memoryuse*, the maximum amount of physical memory a process may have allocated to it at a given time.

The *maximum-use* may be given as a (floating point or integer) number followed by a scale factor. For all limits other than *cputime* the default scale is 'k' or 'kilobytes' (1024 bytes); a scale factor of 'm' or 'megabytes' may also be used. For *cputime* the default scaling is 'seconds', while 'm' for minutes or 'h' for hours, or a time of the form 'mm:ss' giving minutes and seconds may be used.

For both *resource* names and scale factors, unambiguous prefixes of the names suffice.

login

Terminate a login shell, replacing it with an instance of */bin/login*. This is one way to log off, included for compatibility with *sh* (1).

logout

Terminate a login shell. Especially useful if *ignoreeof* is set.

nice
nice +number
nice command
nice +number command

The first form sets the *nice* for this shell to 4. The second form sets the *nice* to the given number. The final two forms run command at priority 4 and *number* respectively. The super-user may specify negative niceness by using **nice** *−number* .... Command is always executed in a sub-shell, and the restrictions place on commands in simple *if* statements apply.

nohup
nohup command

The first form can be used in shell scripts to cause hangups to be ignored for the remainder of the script. The second form causes the specified command to be run with hangups ignored. All processes detached with & are effectively *nohup* ed.

notify
notify %job ...

Causes the shell to notify the user asynchronously when the status of the current or specified jobs changes; normally notification is presented before a prompt. This is automatic if the shell variable *notify* is set.

onintr
onintr −
onintr label

Control the action of the shell on interrupts. The first form restores the default action of the shell on interrupts which is to terminate shell scripts or to return to the terminal command input level. The second form **onintr** − causes all interrupts to be ignored. The final form causes the shell to execute a **goto label** when an interrupt is received or a child process terminates because it was interrupted.

In any case, if the shell is running detached and interrupts are being ignored, all forms of *onintr* have no meaning and interrupts continue to be ignored by the shell and all invoked commands.

popd
popd +n

Pops the directory stack, returning to the new top directory. With an argument '+*n*' discards the *n* th entry in the stack. The elements of the directory stack are numbered from 0 starting at the top.

pushd
pushd name
pushd +n

With no arguments, *pushd* exchanges the top two elements of the directory stack. Given a *name* argument, *pushd* changes to the new directory (ala *cd)* and pushes the old current working directory (as in *csw)* onto the directory stack. With a numeric argument, rotates the *n* th argument of the directory stack around to be the top element and changes to it. The members of the directory stack are numbered from the top starting at 0.

rehash

Causes the internal hash table of the contents of the directories in the *path* variable to be recomputed. This is needed if new commands are added to directories in the *path* while you are logged in. This should only be necessary if you add commands to one of your own directories, or if a systems programmer changes the contents of one of the system directories.

repeat count command

The specified *command* which is subject to the same restrictions as the *command* in the one line *if* statement above, is executed *count* times. I/O redirections occur exactly once, even if *count* is 0.

set
set name
set name=word
set name[index]=word
set name=(wordlist)

The first form of the command shows the value of all shell variables. Variables which have other than a single word as value print as a parenthesized word list. The second form sets *name* to the null string. The third form sets *name* to the single *word*. The fourth form sets the *indexth* component of name to word; this component must already exist. The final form sets *name* to the list of words in *wordlist*. In all cases the value is command and filename expanded.

These arguments may be repeated to set multiple values in a single set command. Note, however, that variable expansion happens for all arguments before any setting occurs.

setenv
setenv name
setenv name value

The first form lists all current environment variables. The last form sets the value of environment variable *name* to be *value,* a single string. The second form sets *name* to an empty string. The variables PATH, USER, LOGNAME, HOME, and TERM are automatically imported to and exported from the *csh* variables *path*, *user*, *logname*, *home*, and *term*, respectively; there is no need to use *setenv* for these.

shift
shift variable

The members of *argv* are shifted to the left, discarding *argv*[1]. It is an error for *argv* not to be set or to have less than one word as value. The second form performs the same function on the specified variable.

source name
source –**h** name

The shell reads commands from *name*. *Source* commands may be nested; if they are nested too deeply the shell may run out of file descriptors. An error in a *source* at any level terminates all nested *source* commands. Normally input during *source* commands is not placed on the history list; the –h option causes the commands to be placed in the history list without being executed.

stop
stop %job ...

Stops the current or specified job which is executing in the background.

suspend

Causes the shell to stop in its tracks, much as if it had been sent a stop signal with ^Z. This is most often used to stop shells started by *su*(1).

switch (string)
case str1:
    ...
        breaksw
...
default:

...

breaksw

endsw

Each case label is successively matched against the specified *string* which is first command and filename expanded. The file metacharacters *, ? and [...] may be used in the case labels, which are variable expanded. If none of the labels match before a default label is found, then the execution begins after the default label. Each case label and the default label must appear at the beginning of a line. The command *breaksw* causes execution to continue after the *endsw*. Otherwise control may fall through case labels and default labels as in C. If no label matches and there is no default, execution continues after the *endsw*.

time
time command

With no argument, a summary of time used by this shell and its children is printed. If arguments are given, the specified simple command is timed and a time summary as described under the *time* variable is printed. If necessary, an extra shell is created to print the time statistic when the command completes.

umask
umask value

The file creation mask is displayed (first form) or set to the specified value (second form). The mask is given in octal. Common values for the mask are 002 giving all access to the group and read and execute access to others or 022 giving all access except no write access for users in the group or others.

unalias pattern

All aliases whose names match the specified pattern are discarded. Thus all aliases are removed by **unalias** *. It is not an error for nothing to be *unaliased*.

unhash

Use of the internal hash table to speed location of executed programs is disabled.

unlimit
unlimit *resource*
unlimit –h
unlimit –h *resource*

Removes the limitation on *resource*. If no *resource* is specified, then all *resource* limitations are removed. If –h is given, the corresponding hard limits are removed. Only the super-user may do this.

unset pattern

All variables whose names match the specified pattern are removed. Thus all variables are removed by **unset** *; this has noticeably distasteful side-effects. It is not an error for nothing to be *unset*.

unsetenv pattern

Removes all variables whose name match the specified pattern from the environment. See also the *setenv* command above and *env*(1).

wait

All background jobs are waited for. If the shell is interactive, then an interrupt can disrupt the wait, at which time the shell prints names and job numbers of all jobs known to be outstanding.

while (expr)
...
end

While the specified expression evaluates non-zero, the commands between the *while* and the matching end are evaluated. *Break* and *continue* may be used to terminate or continue the loop prematurely. (The *while* and *end* must appear alone on their input lines.) Prompting occurs here the first time through the loop as for the *foreach* statement if the input is a terminal.

%job

Brings the specified job into the foreground.

%job &

Continues the specified job in the background.

@
@ name = expr
@ name[index] = expr

The first form prints the values of all the shell variables. The second form sets the specified *name* to the value of *expr*. If the expression contains <, >, & or |, then at least this part of the expression must be placed within ( ). The third form assigns the value of *expr* to the *index*th argument of *name*. Both *name* and its *index*th component must already exist.

The operators *=, +=, etc., are available as in C. The space separating the name from the assignment operator is optional. Spaces are, however, mandatory in separating components of *expr* which would otherwise be single words.

Special postfix ++ and -- operators increment and decrement *name* respectively, i.e., @ i++.

Pre-defined and Environment Variables

The following variables have special meaning to the shell. Of these, *argv*, *cwd*, *home*, *path*, *prompt*, *shell* and *status* are always set by the shell. Except for *cwd* and *status*, this setting occurs only at initialization; these variables will not then be modified unless this is done explicitly by the user.

This shell copies the environment variable HOME into *home*, and copies it back into the environment whenever the normal shell variables are reset. The environment variable PATH is likewise handled; it is not necessary to worry about its setting other than in the file *.cshrc* as inferior *csh* processes will import the definition of *path* from the environment, and re-export it if you then change it.

argv	Set to the arguments to the shell, it is from this variable that positional parameters are substituted, i.e., $1 is replaced by $argv[1], etc.
cdpath	Gives a list of alternate directories searched to find subdirectories in *chdir* commands.
child	Set by the shell to the process id of the last background job.
cwd	The full pathname of the current directory.
echo	Set when the −x command line option is given. Causes each command and its arguments to be echoed just before it is executed. For non-built-in commands all expansions occur before echoing. Built-in commands are echoed before command and filename substitution, since these substitutions are then done selectively.
fignore	Set to a list of filename suffixes to be ignored when doing file name completion.
filec	Enable file name completion. Note that setting **filec** causes the end-of-line character for the terminal (the EOL character described in *termio*(7)) to become undefined.
histchars	Can be given a string value to change the characters used in history substitution. The first character of its value is used as the history substitution character, replacing the default character !. The second character of its value replaces the character ↑ in quick substitutions.
history	Can be given a numeric value to control the size of the history list. Any command which has been referenced in this many events will not be discarded. Too large values of *history* may run the shell out of memory. The last executed command is always saved on the history list.

home The home directory of the invoker, initialized from the environment. The filename expansion of ˜ refers to this variable.

ignoreeof If set the shell ignores end-of-file from input devices which are terminals. This prevents shells from accidentally being killed by typing the EOF character (control-D by default).

mail The files where the shell checks for mail. This is done after each command completion which will result in a prompt, if a specified interval has elapsed. If the file exists with an access time not greater than its modify time, the shell says "You have new mail.".

If the first word of the value of *mail* is numeric, it specifies a different mail checking interval, in seconds, than the default, which is 10 minutes.

If multiple mail files are specified, then the shell says "New mail in *name*" when there is mail in the file *name*.

nobeep Disables the sounding of the terminal bell when there are multiple possible file name completions.

noclobber As described in the section on Input/output, restrictions are placed on output redirection to insure that files are not accidentally destroyed, and that >> redirections refer to existing files.

noglob If set, filename expansion is inhibited. This is most useful in shell scripts which are not dealing with filenames, or after a list of filenames has been obtained and further expansions are not desirable.

nonomatch If set, it is not an error for a filename expansion to not match any existing files; rather the primitive pattern is returned. It is still an error for the primitive pattern to be malformed, i.e., echo [still gives an error.

notify If set, the shell notifies asynchronously of job completions. The default is to present job completions just before printing a prompt.

path Each word of the path variable specifies a directory in which commands are to be sought for execution. A null word specifies the current directory. If there is no *path* variable, then only full path names will execute. The default search path for users other than the super user is ".", */usr/sbin*, */usr/bsd*, */bin*, */usr/bin* and */usr/bin/X11*. For the super-user the default search path is */usr/sbin, /usr/bsd, /bin, /usr/bin,*

/etc, /usr/etc, /usr/bin/X11. A shell which is given neither the **–c** nor the **–t** option will normally hash the contents of the directories in the *path* variable after reading *.cshrc*, and each time the *path* variable is reset. If new commands are added to these directories while the shell is active, it may be necessary to give the *rehash* or the commands may not be found.

prompt The string which is printed before each command is read from an interactive terminal input. If a ! appears in the string, it will be replaced by the current event number unless a preceding \ is given. The sequence \\ is replaced with a single \. The prompt should only be set by the user if it is already defined so that it will not be printed when processing shell scripts by using the statement

```
if ( $?prompt ) set prompt='\!% '
```

If the sequence \@x appears, where x is one of the characters listed below, then it will be replaced by the current time and date in the indicated format.

R	time as HH:MM AM/PM, e.g., 8:40PM
r	time as HH:MM:SS AM/PM, e.g., 08:40:25 PM
m	month of year – 01 to 12
d	day of month – 01 to 31
y	last 2 digits of year – 00 to 99
D	date as mm/dd/yy
H	hour – 00 to 23
M	minute – 00 to 59
S	second – 00 to 59
T	time as HH:MM:SS
j	day of year – 001 to 366
w	day of week – Sunday = 0
a	abbreviated weekday – Sun to Sat
h	abbreviated month – Jan to Dec
n	insert a new-line character
t	insert a tab character

The default prompt is %, or # for the super-user.

savehist is given a numeric value to control the number of entries of the history list that are saved in *~/.history* when the user logs out. The most recent *$savehist* commands will be saved. During start up the shell sources *~/.history* into the history list, thus enabling history to be saved across login sessions. Large values of *savehist* will slow down the shell during start up. To prevent *su* sessions from clobbering the underlying user's history file, the shell will only write in the *~/.history* file if its current effective user id is the same as the owner of the directory specified by *$home*.

shell The file in which the shell resides. This is used in forking shells to interpret files which have execute bits set, but which are not executable by the system. (See the description of Non-built-in Command Execution below.) Initialized to the (system-dependent) home of the shell.

status The status returned by the last command. If it terminated abnormally, then 0200 is added to the status. Built-in commands which fail return exit status 1, all other built-in commands set status 0.

time Controls automatic timing of commands. If set, then any command which takes more than this many cpu seconds will cause a line giving user, system, and real times and a utilization percentage which is the ratio of user plus system times to real time to be printed when it terminates.

verbose Set by the **−v** command line option, causes the words of each command to be printed after history substitution.

Non-built-in Command Execution

When a command to be executed is found not to be a built-in command, the shell attempts to execute the command via *exec* (2). Each word in the variable *path* names a directory from which the shell will attempt to execute the command. If it is given neither a **−c** nor a **−t** option, the shell will hash the names in these directories into an internal table so that it will only try an *exec* in a directory if there is a possibility that the command resides there. This greatly speeds command location when a large number of directories are present in the search path. If this mechanism has been turned off (via *unhash*), or if the shell was given a **−c** or **−t** argument, and in any case for each directory component of *path* which does not begin with a /, the shell concatenates with the given command name to form a path name of a file which it then attempts to execute.

Parenthesized commands are always executed in a subshell. Thus **(cd ; pwd) ; pwd** prints the *home* directory; leaving you where you were (printing this after the home directory), while **cd ; pwd** leaves you in the *home* directory. Parenthesized commands are most often used to prevent *chdir* from affecting the current shell.

If the file has execute permissions but is not an executable binary to the system, then it is assumed to be a file containing shell commands an a new shell is spawned to read it.

If there is an *alias* for *shell*, then the words of the alias will be prepended to the argument list to form the shell command. The first word of the *alias* should be the full path name of the shell (e.g., "$shell"). Note that this is a special, late occurring, case of *alias* substitution, and only allows words to be prepended to the argument list without modification.

Argument List Processing

If argument 0 to the shell is –, then this is a login shell. The flag arguments are interpreted as follows:

–b This flag forces a ''break'' from option processing, causing any further shell arguments to be treated as non-option arguments. The remaining arguments will not be interpreted as shell options. This may be used to pass options to a shell script without confusion or possible subterfuge. The shell will not run a set-user ID script without this option.

–c Commands are read from the (single) following argument which must be present. Any remaining arguments are placed in *argv*.

–e The shell exits if any invoked command terminates abnormally or yields a non-zero exit status.

–f The shell will start faster, because it will neither search for nor execute commands from the file *.cshrc* in the invokers home directory.

–i The shell is interactive and prompts for its top-level input, even if it appears to not be a terminal. Shells are interactive without this option if their inputs and outputs are terminals.

–n Commands are parsed, but not executed. This may aid in syntactic checking of shell scripts.

–s Command input is taken from the standard input.

–t A single line of input is read and executed. A \ may be used to escape the newline at the end of this line and continue onto another line.

–v Causes the *verbose* variable to be set, with the effect that command input is echoed after history substitution.

–x Causes the *echo* variable to be set, so that commands are echoed immediately before execution.

–V Causes the *verbose* variable to be set even before *.cshrc* is executed.

–X Is to –x as –V is to –v.

After processing of flag arguments, if arguments remain but none of the –c, –i, –s, or –t options was given, the first argument is taken as the name of a file of commands to be executed. The shell opens this file, and saves its name for possible resubstitution by $0. Remaining arguments initialize the variable *argv*. *csh* scripts should always start with

```
#! /bin/csh -f
```

which causes the kernel to fork off */bin/csh* to process them even if invoked by a Bourne shell user and inhibits processing of the *.cshrc* file to prevent interference by the user's differing aliases.

Signal Handling

The shell normally ignores *quit* signals. Processes running in background (either by '&' or the *bg* or *%... &* commands) are immune to signals generated from the keyboard, namely, *interrupt* and *quit*, and to hangups. Other signals have the values which the shell inherited from its parent. The handling of interrupts and terminate signals in shell scripts can be controlled by *onintr*. Login shells catch the *terminate* signal; otherwise this signal is passed on to children from the state in the shell's parent. In no case are interrupts allowed when a login shell is reading the file *~/.logout*.

EXAMPLE

```
csh
```

creates a new C shell which will accept shell commands.

FILES

/etc/cshrc	Read during initialization by login shells only.
~/.cshrc	Read at beginning of execution by each shell (after */etc/cshrc*, if applicable).
~/.login	Read by login shell, after *.cshrc* at login.
~/.logout	Read by login shell, at logout.
/bin/sh	Standard shell, for shell scripts not starting with a #.
/tmp/sh*	Temporary file for <<.
/etc/passwd	Source of home directories for ~name.

LIMITATIONS

Words can be no longer than 1024 characters. The system limits argument lists to 10240 characters. The number of arguments to a command which involves filename expansion is limited to 1/6th the number of characters allowed in an argument list. Command substitutions may substitute no more characters than are allowed in an argument list. To detect looping, the shell restricts the number of *alias* substitutions on a single line to 20.

SEE ALSO

sh(1), access(2), exec(2), fork(2), getrlimit(2), pipe(2), signal(2), umask(2), wait(2), a.out(4), environ(5), termio(7).
An Introduction to the C Shell, by William Joy.

BUGS

When a command is restarted from a stop, the shell prints the directory it started in if this is different from the current directory; this can be misleading (i.e., wrong) as the job may have changed directories internally.

Shell builtin functions are not stoppable/restartable. Command sequences of the form 'a ; b ; c' are also not handled gracefully when stopping is attempted. If you suspend 'b', the shell will then immediately execute 'c'. This is especially noticeable if this expansion results from an *alias*. It suffices to place the sequence of commands in ()'s to force it to a subshell, i.e., '(a ; b ; c)'.

Control over tty output after processes are started is primitive; perhaps this will inspire someone to work on a good virtual terminal interface. In a virtual terminal interface much more interesting things could be done with output control.

Alias substitution is most often used to clumsily simulate shell procedures; shell procedures should be provided rather than aliases.

Commands within loops, prompted for by '?', are not placed in the *history* list. Control structures should be parsed rather than being recognized as built-in commands. This would allow control commands to be placed anywhere, to be combined with '|', and to be used with '&' and ';' metasyntax.

It should be possible to use the ':' modifiers on the output of command substitutions. All and more than one ':' modifier should be allowed on '$' substitutions.

The way the **filec** facility is implemented is ugly and expensive.

Bourne shell scripts which start with # will be executed by *csh* unless they use the kernel's #! facility, e.g.,

```
#! /bin/sh
```

NAME

csplit – context split

SYNOPSIS

csplit [–s] [–k] [–f prefix] file arg1 [... argn]

DESCRIPTION

csplit reads *file* and separates it into n+1 sections, defined by the arguments *arg1 ... argn*. By default the sections are placed in xx00 ... xx*n* (*n* may not be greater than 99). These sections get the following pieces of *file*:

00: From the start of *file* up to (but not including) the line referenced by *arg1*.

01: From the line referenced by *arg1* up to the line referenced by *arg2*.

•
•
•

n+1: From the line referenced by *argn* to the end of *file*.

If the *file* argument is a – then standard input is used.

The options to *csplit* are:

–s *csplit* normally prints the character counts for each file created. If the –s option is present, *csplit* suppresses the printing of all character counts.

–k *csplit* normally removes created files if an error occurs. If the **–k** option is present, *csplit* leaves previously created files intact.

–f *prefix* If the **–f** option is used, the created files are named *prefix*00 ... *prefixn*. The default is xx00 ... xx*n*.

The arguments (*arg1 ... argn*) to *csplit* can be a combination of the following:

/rexp/ A file is to be created for the section from the current line up to (but not including) the line containing the regular expression *rexp*. The current line becomes the line containing *rexp*. This argument may be followed by an optional + or – some number of lines (e.g., **/Page/–5**).

%rexp% This argument is the same as */rexp/*, except that no file is created for the section.

lnno A file is to be created from the current line up to (but not including) *lnno*. The current line becomes *lnno*.

{*num*} Repeat argument. This argument may follow any of the above arguments. If it follows a *rexp* type argument, that argument is applied *num* more times. If it follows *lnno*, the file will be split every *lnno* lines (*num* times) from that point.

Enclose all *rexp* type arguments that contain blanks or other characters meaningful to the shell in the appropriate quotes. Regular expressions may not contain embedded new-lines. *csplit* does not affect the original file; it is the users responsibility to remove it.

EXAMPLES

```
csplit –f cobol file '/procedure division/' /par5./ /par16./
```

This example creates four files, **cobol00** . . . **cobol03**. After editing the ''split'' files, they can be recombined as follows:

```
cat cobol0[0–3] > file
```

Note that this example overwrites the original file.

```
csplit –k file 100 {99}
```

This example would split the file at every 100 lines, up to 10,000 lines. The **–k** option causes the created files to be retained if there are less than 10,000 lines; however, an error message would still be printed.

```
csplit –k prog.c '%main(%' '/^}/+1' {20}
```

Assuming that **prog.c** follows the normal C coding convention of ending routines with a } at the beginning of the line, this example will create a file containing each separate C routine (up to 21) in **prog.c**. Lines before the line containing `main(` are not copied to any file. The error message

{20} – out of range

indicates (if it appears) there were fewer than 21 lines ending with } so fewer than 21 files were created.

SEE ALSO

ed(1), sh(1).
regexp(5) in the *Programmer's Reference Manual.*

DIAGNOSTICS

Self-explanatory except for:

arg – out of range

which means that the given argument did not reference a line between the current position and the end of the file.

NAME

ct – spawn getty to a remote terminal

SYNOPSIS

ct [**–wn**] [**–xn**] [**–h**] [**–v**] [**–s**speed] telno ...

DESCRIPTION

ct dials the telephone number of a modem that is attached to a terminal, and spawns a *getty* process to that terminal. *Telno* is a telephone number, with equal signs for secondary dial tones and minus signs for delays at appropriate places. (The set of legal characters for *telno* is 0 thru 9, -, =, *, and #. The maximum length *telno* is 31 characters). If more than one telephone number is specified, *ct* will try each in succession until one answers; this is useful for specifying alternate dialing paths.

ct will try each line listed in the file **/usr/lib/uucp/Devices** until it finds an available line with appropriate attributes or runs out of entries. If there are no free lines, *ct* will ask if it should wait for one, and if so, for how many minutes it should wait before it gives up. *ct* will continue to try to open the dialers at one-minute intervals until the specified limit is exceeded. The dialogue may be overridden by specifying the **–w***n* option, where *n* is the maximum number of minutes that *ct* is to wait for a line.

The **–x***n* option is used for debugging; it produces a detailed output of the program execution on stderr. The debugging level, *n*, is a single digit; **–x9** is the most useful value.

Normally, *ct* will hang up the current line, so the line can answer the incoming call. The **–h** option will prevent this action. The **–h** option will also wait for the termination of the specified *ct* process before returning control to the user's terminal. If the **–v** option is used, *ct* will send a running narrative to the standard error output stream.

The data rate may be set with the **–s** option, where *speed* is expressed in baud. The default rate is 1200.

After the user on the destination terminal logs out, there are two things that could occur depending on what type of getty is on the line (*getty* or *uugetty*). For the first case, *ct* prompts, **Reconnect?** If the response begins with the letter **n**, the line will be dropped; otherwise, *getty* will be started again and the **login:** prompt will be printed. In the second case, there is already a getty (*uugetty*) on the line, so the **login:** message will appear.

To log out properly, the user must type **control D**.

Of course, the destination terminal must be attached to a modem that can answer the telephone.

FILES

/usr/lib/uucp/Devices
/usr/adm/ctlog

SEE ALSO

cu(1C), login(1), uucp(1C).
getty(1M), uugetty(1M) in the *System Administrator's Reference Manual.*

BUGS

For a shared port, one used for both dial-in and dial-out, the *uugetty* program running on the line must have the –r option specified (see *uugetty*(1M)).

NAME

ctags – create a tags file

SYNOPSIS

ctags [**–BFatuwvx**] [**–f** *tagsfile*] name ...

DESCRIPTION

Ctags makes a tags file for *ex*(1) from the specified C, Pascal, Fortran, YACC, lex, and lisp sources. A tags file gives the locations of specified objects (in this case functions and typedefs) in a group of files. Each line of the tags file contains the object name, the file in which it is defined, and an address specification for the object definition. Functions are searched with a pattern, typedefs with a line number. Specifiers are given in separate fields on the line, separated by blanks or tabs. Using the *tags* file, *ex* can quickly find these objects definitions.

If the **–x** flag is given, *ctags* produces a list of object names, the line number and file name on which each is defined, as well as the text of that line and prints this on the standard output. This is a simple index which can be printed out as an off-line readable function index.

Normally *ctags* places the tag descriptions in a file called *tags*; this may be overridden with the **–f** option.

Files whose names end in **.c** or **.h** are assumed to be C source files and are searched for C routine and macro definitions. Files whose names end in **.y** are assumed to be YACC source files. Files whose names end in **.l** are assumed to be either lisp files if their first non-blank character is ';', '(', or '[', or lex files otherwise. Other files are first examined to see if they contain any Pascal or Fortran routine definitions; if not, they are processed again looking for C definitions.

Other options are:

–F use forward searching patterns (/.../) (default).

–B use backward searching patterns (?...?).

–a append to tags file. Since *ex* and *vi* use a binary search to locate tags, the resulting tags file should be sorted using *sort*(1).

–t create tags for typedefs.

–v Produce on the standard output an index of the form expected by **vgrind**(1). This listing contains the function name, file name, and page number (assuming 64 line pages).

–w suppressing warning diagnostics.

−u causing the specified files to be *updated* in tags, that is, all references to them are deleted, and the new values are appended to the file. (Beware: this option is implemented in a way which is rather slow; it is usually faster to simply rebuild the *tags* file.)

The tag *main* is treated specially in C programs. The tag formed is created by prepending *M* to the name of the file, with a trailing .c removed, if any, and leading pathname components also removed. This makes use of *ctags* practical in directories with more than one program.

FILES

tags	output tags file

SEE ALSO

ex(1), vi(1)

BUGS

Recognition of **functions, subroutines** and **procedures** for FORTRAN and Pascal is done is a very simpleminded way. No attempt is made to deal with block structure; if you have two Pascal procedures in different blocks with the same name you lose.

The method of deciding whether to look for C or Pascal and FORTRAN functions is a hack.

Does not know about #ifdefs.

Should know about Pascal types.

Use of -tx shows only the last line of typedefs. Relies on the input being well formed to detect typedefs. Typedefs are only partly understood. **typedef const unsigned short int Z;** generates output which claims **unsigned**, **short, int**, and Z are all typedefs.

NAME

ctrace – C program debugger

SYNOPSIS

ctrace [options] [file]

DESCRIPTION

The *ctrace* command allows you to follow the execution of a C program, statement-by-statement. The effect is similar to executing a shell procedure with the **–x** option. *ctrace* reads the C program in *file* (or from standard input if you do not specify *file*), inserts statements to print the text of each executable statement and the values of all variables referenced or modified, and writes the modified program to the standard output. You must put the output of *ctrace* into a temporary file because the *cc*(1) command does not allow the use of a pipe. You then compile and execute this file.

As each statement in the program executes it will be listed at the terminal, followed by the name and value of any variables referenced or modified in the statement, followed by any output from the statement. Loops in the trace output are detected and tracing is stopped until the loop is exited or a different sequence of statements within the loop is executed. A warning message is printed every 1000 times through the loop to help you detect infinite loops. The trace output goes to the standard output so you can put it into a file for examination with an editor or the *bfs*(1) or *tail*(1) commands.

The options commonly used are:

–f *functions*	Trace only these *functions.*
–v *functions*	Trace all but these *functions.*

You may want to add to the default formats for printing variables. Long and pointer variables are always printed as signed integers. Pointers to character arrays are also printed as strings if appropriate. Char, short, and int variables are also printed as signed integers and, if appropriate, as characters. Double variables are printed as floating point numbers in scientific notation. You can request that variables be printed in additional formats, if appropriate, with these options:

–o	Octal
–x	Hexadecimal
–u	Unsigned
–e	Floating point

These options are used only in special circumstances:

–l *n*	Check *n* consecutively executed statements for looping trace output, instead of the default of 20. Use 0 to get all the trace output from loops.

–s Suppress redundant trace output from simple assignment statements and string copy function calls. This option can hide a bug caused by use of the = operator in place of the == operator.

–t *n* Trace *n* variables per statement instead of the default of 10 (the maximum number is 20). The Diagnostics section explains when to use this option.

–P Run the C preprocessor on the input before tracing it. You can also use the **–D**, **–I**, and **–U** *cpp*(1) options.

These options are used to tailor the run-time trace package when the traced program will run in a non-UNIX System environment:

–b Use only basic functions in the trace code, that is, those in *ctype*(3C), *printf*(3S), and *string*(3C). These are usually available even in cross-compilers for microprocessors. In particular, this option is needed when the traced program runs under an operating system that does not have *signal*(2), *fflush*(3S), *longjmp*(3C), or *setjmp*(3C).

–p *string*
Change the trace print function from the default of 'printf('. For example, 'fprintf(stderr,' would send the trace to the standard error output.

–r *f* Use file *f* in place of the *runtime.c* trace function package. This lets you change the entire print function, instead of just the name and leading arguments (see the **–p** option).

EXAMPLE

If the file *lc.c* contains this C program:

```
#include <stdio.h>
main()          /* count lines in input */
{
        int c, nl;

        nl = 0;
        while ((c = getchar()) != EOF)
                if (c = '\n')
                        ++nl;
        printf("%d\n", nl);
}
```

and you enter these commands and test data:

```
cc lc.c
a.out
1
(cntl-d)
```

the program will be compiled and executed. The output of the program will be the number 2, which is not correct because there is only one line in the test data. The error in this program is common, but subtle. If you invoke *ctrace* with these commands:

```
        ctrace lc.c >temp.c
        cc temp.c
        a.out
```

the output will be:

```
         2 main()
         6        nl = 0;
                  /* nl == 0 */
         7        while ((c = getchar()) != EOF)
```

The program is now waiting for input. If you enter the same test data as before, the output will be:

```
                  /* c == 49 or '1' */
         8                if (c = '\n')
                          /* c == 10 or '\n' */
         9                        ++nl;
                                  /* nl == 1 */
         7        while ((c = getchar()) != EOF)
                  /* c == 10 or '\n' */
         8                if (c = '\n')
                          /* c == 10 or '\n' */
         9                        ++nl;
                                  /* nl == 2 */
         7        while ((c = getchar()) != EOF)
```

If you now enter an end of file character (cntl-d) the final output will be:

```
                  /* c == -1 */
         10       printf("%d\n", nl);
                  /* nl == 2 */2
                   return
```

Note that the program output printed at the end of the trace line for the **nl** variable. Also note the **return** comment added by *ctrace* at the end of the trace output. This shows the implicit return at the terminating brace in the function.

The trace output shows that variable **c** is assigned the value '1' in line 7, but in line 8 it has the value '\n'. Once your attention is drawn to this **if** statement, you will probably realize that you used the assignment operator (=) in place of the equality operator (==). You can easily miss this error during code reading.

EXECUTION-TIME TRACE CONTROL

The default operation for *ctrace* is to trace the entire program file, unless you use the –f or –v options to trace specific functions. This does not give you statement-by-statement control of the tracing, nor does it let you turn the tracing off and on when executing the traced program.

You can do both of these by adding *ctroff*() and *ctron*() function calls to your program to turn the tracing off and on, respectively, at execution time. Thus, you can code arbitrarily complex criteria for trace control with *if* statements, and you can even conditionally include this code because *ctrace* defines the **CTRACE** preprocessor variable. For example:

```
#ifdef CTRACE
        if (c == '!' && i > 1000)
                ctron();
#endif
```

You can also control tracing from *edge*(1) or *dbx*(1) if you compile with the –g option by turning trace on and off with the static variable tr_ct_.. For example, to trace all but lines 7 to 10 in the main function, enter:

```
dbx a.out
when at 7 { assign tr_ct_ = 0 }
when at 11 { assign tr_ct_ = 1 }
run
```

DIAGNOSTICS

This section contains diagnostic messages from both *ctrace* and *cc*(1), since the traced code often gets some *cc* warning messages. You can get *cc* error messages in some rare cases, all of which can be avoided.

ctrace Diagnostics

warning: some variables are not traced in this statement

Only 10 variables are traced in a statement to prevent the C compiler "out of tree space; simplify expression" error. Use the –t option to increase this number.

warning: statement too long to trace

This statement is over 400 characters long. Make sure that you are using tabs to indent your code, not spaces.

cannot handle preprocessor code, use –P option

This is usually caused by #ifdef/#endif preprocessor statements in the middle of a C statement, or by a semicolon at the end of a #define preprocessor statement.

'if ... else if' sequence too long
: Split the sequence by removing an else from the middle.

possible syntax error, try –P option
: Use the **–P** option to preprocess the *ctrace* input, along with any appropriate **–D**, **–I**, and **–U** preprocessor options. If you still get the error message, check the Warnings section below.

Cc Diagnostics

warning: illegal combination of pointer and integer
warning: statement not reached
warning: sizeof returns 0
: Ignore these messages.

compiler takes size of function
: See the *ctrace* "possible syntax error" message above.

yacc stack overflow
: See the *ctrace* "'if ... else if' sequence too long" message above.

out of tree space; simplify expression
: Use the **–t** option to reduce the number of traced variables per statement from the default of 10. Ignore the "ctrace: too many variables to trace" warnings you will now get.

redeclaration of signal
: Either correct this declaration of *signal*(2), or remove it and #include <signal.h>.

SEE ALSO

bfs(1), edge(1), dbx(1), tail(1), signal(2), ctype(3C), fclose(3S), printf(3S), setjmp(3C), string(3C).

WARNINGS

You will get a *ctrace* syntax error if you omit the semicolon at the end of the last element declaration in a structure or union, just before the right brace (}). This is optional in some C compilers.

Defining a function with the same name as a system function may cause a syntax error if the number of arguments is changed. Just use a different name.

ctrace assumes that BADMAG is a preprocessor macro, and that EOF and NULL are #defined constants. Declaring any of these to be variables, e.g., "int EOF;", will cause a syntax error.

BUGS

ctrace does not know about the components of aggregates like structures, unions, and arrays. It cannot choose a format to print all the components of an aggregate when an assignment is made to the entire aggregate. *ctrace* may choose to print the address of an aggregate or use the wrong format (e.g., 3.149050e-311 for a structure with two integer members) when printing the value of an aggregate.

Pointer values are always treated as pointers to character strings.

The loop trace output elimination is done separately for each file of a multi-file program. This can result in functions called from a loop still being traced, or the elimination of trace output from one function in a file until another in the same file is called.

FILES

/usr/lib/ctrace/runtime.c	run-time trace package

NAME

cu – call another UNIX system

SYNOPSIS

cu [**–s**speed] [**–l**line] [**–h**] [**–t**] [**–d**] [**–o** | **–e**] [**–n**] telno
cu [**–s** speed] [**–h**] [**–d**] [**–o** | **–e**] **–l** line
cu [**–h**] [**–d**] [**–o** | **–e**] systemname

DESCRIPTION

cu calls up another UNIX system, a terminal, or possibly a non-UNIX system. It manages an interactive conversation with possible transfers of ASCII files.

cu accepts the following options and arguments:

–s*speed* Specifies the transmission speed (300, 1200, 2400, 4800, 9600); The default value is "Any" speed which will depend on the order of the lines in the **/usr/lib/uucp/Devices** file.

–l*line* Specifies a device name to use as the communication line. This can be used to override the search that would otherwise take place for the first available line having the right speed. When the –l option is used without the –s option, the speed of a line is taken from the Devices file. When the –l and –s options are both used together, cu will search the Devices file to check if the requested speed for the requested line is available. If so, the connection will be made at the requested speed; otherwise an error message will be printed and the call will not be made. The specified device is generally a directly connected asynchronous line (e.g., **/dev/ttyd4**) in which case a telephone number (*telno*) is not required. The specified device need not be in the **/dev** directory. If the specified device is associated with an auto dialer, a telephone number must be provided.

A line also used by *uugetty*(1M) cannot be used with this option. If *uugetty*(1M) is being used, there is likely to be an autodialing modem on the line, using an entry in the Dialers file like the **Hayes24** entry, and accepting a telephone, *telno*, number from the *cu* command.

Use of this option with *systemname* rather than *telno* will not give the desired result (see *systemname* below).

−h	Emulates local echo, supporting calls to other computer systems which expect terminals to be set to half-duplex mode.
−t	Used to dial an ASCII terminal which has been set to auto answer. Appropriate mapping of carriage-return to carriage-return-line-feed pairs is set.
−d	Causes diagnostic traces to be printed.
−o	Designates that odd parity is to be generated for data sent to the remote system.
−n	For added security, will prompt the user to provide the telephone number to be dialed rather than taking it from the command line.
−e	Designates that even parity is to be generated for data sent to the remote system.
telno	When using an automatic dialer, the argument is the telephone number with equal signs for secondary dial tone or minus signs placed appropriately for delays of 4 seconds.
systemname	A uucp system name may be used rather than a telephone number; in this case, *cu* will obtain an appropriate direct line or telephone number from **/usr/lib/uucp/Systems.** Note: the *systemname* option should not be used in conjunction with the **−l** and **−s** options as *cu* will connect to the first available line for the system name specified, ignoring the requested line and speed.

After making the connection, *cu* runs as two processes: the *transmit* process reads data from the standard input and, except for lines beginning with ˜, passes it to the remote system; the *receive* process accepts data from the remote system and, except for lines beginning with ˜, passes it to the standard output. Normally, an automatic DC3/DC1 protocol is used to control input from the remote so the buffer is not overrun. Lines beginning with ˜ have special meanings.

The *transmit* process interprets the following user initiated commands:

˜**.**	terminate the conversation.
˜**!**	escape to an interactive shell on the local system.
˜**!***cmd*...	run *cmd* on the local system (via **sh −c**).
˜**$***cmd*...	run *cmd* locally and send its output to the remote system.

~^Z	suspend the *cu* session. (^Z, control-Z, is the current job control suspend character (see *csh*(1) and *stty*(1)).
~%cd	change the directory on the local system. Note: **~!cd** will cause the command to be run by a sub-shell, probably not what was intended.
~%take *from* [*to*]	copy file *from* (on the remote system) to file *to* on the local system. If *to* is omitted, the *from* argument is used in both places.
~%put *from* [*to*]	copy file *from* (on local system) to file *to* on remote system. If *to* is omitted, the *from* argument is used in both places. For both **~%take** and **put** commands, as each block of the file is transferred, consecutive single digits are printed to the terminal.
~~ *line*	send the line ~ *line* to the remote system.
~%break	transmit a **BREAK** to the remote system (which can also be specified as **~%b**).
~%debug	toggles the **-d** debugging option on or off (which can also be specified as **~%d**).
~t	prints the values of the termio structure variables for the user's terminal (useful for debugging).
~l	prints the values of the termio structure variables for the remote communication line (useful for debugging).
~%nostop	toggles between DC3/DC1 input control protocol and no input control. This is useful in case the remote system is one which does not respond properly to the DC3 and DC1 characters.

The *receive* process normally copies data from the remote system to its standard output. Internally the program accomplishes this by initiating an output diversion to a file when a line from the remote begins with ~.

Data from the remote is diverted (or appended, if >> is used) to *file* on the local system. The trailing ~> marks the end of the diversion.

The use of **~%put** requires *stty*(1) and *cat*(1) on the remote side. It also requires that the current erase and kill characters on the remote system be identical to these current control characters on the local system. Backslashes are inserted at appropriate places.

The use of ~%**take** requires the existence of *echo*(1) and *cat*(1) on the remote system. Also, *tabs* mode (See *stty(1))* should be set on the remote system if tabs are to be copied without expansion to spaces.

When *cu* is used on system X to connect to system Y and subsequently used on system Y to connect to system Z, commands on system Y can be executed by using ~~. Executing a tilde command reminds the user of the local system uname. For example, uname can be executed on Z, X, and Y as follows:

```
uname
Z
~[X]!uname
X
~~[Y]!uname
Y
```

In general, ~ causes the command to be executed on the original machine, ~~ causes the command to be executed on the next machine in the chain.

EXAMPLES

To dial a system whose telephone number is 9 201 555 1212 using 1200 baud (where dialtone is expected after the 9):

```
cu –s1200 9=12015551212
```

If the speed is not specified, "Any" is the default value.

To login to a system connected by a direct line:

```
cu –l /dev/ttyXX
```

or

```
cu –l ttyXX
```

To dial a system with the specific line and a specific speed:

```
cu –s1200 –l ttyXX
```

To dial a system using a specific line associated with an auto dialer:

```
cu –l culXX 9=12015551212
```

To use a system name:

```
cu systemname
```

FILES

/usr/lib/uucp/Systems
/usr/lib/uucp/Devices
/usr/spool/locks/LCK..(tty-device)
/dev/tty*

SEE ALSO

cat(1), ct(1C), duart(7), echo(1), stty(1), uucp(1C), uugetty(1M), uname(1).

DIAGNOSTICS

Exit code is zero for normal exit, otherwise, one.

WARNINGS

The *cu* command does not do any integrity checking on data it transfers. Data fields with special *cu* characters may not be transmitted properly. Depending on the interconnection hardware, it may be necessary to use a ~. to terminate the conversion even if **stty 0** has been used. Non-printing characters are not dependably transmitted using either the **~%put** or **~%take** commands. *cu* between an IMBR1 and a penril modem will not return a login prompt immediately upon connection. A carriage return will return the prompt.

The device names, dialers, and so forth in */usr/lib/uucp/Devices* must be correct. Because *cu* is suid-uucp, the device used, such as */dev/ttym3*, must be readable and writable by the "user" uucp.

BUGS

There is an artificial slowing of transmission by *cu* during the **~%put** operation so that loss of data is unlikely.

NAME

cut – cut out selected fields of each line of a file

SYNOPSIS

cut –**c**list [file ...]

cut –**f**list [–**d** char] [–**s**] [file ...]

DESCRIPTION

Use *cut* to cut out columns from a table or fields from each line of a file; in data base parlance, it implements the projection of a relation. The fields as specified by *list* can be fixed length, i.e., character positions as on a punched card (–**c** option) or the length can vary from line to line and be marked with a field delimiter character like *tab* (–**f** option). *cut* can be used as a filter; if no files are given, the standard input is used. In addition, a file name of ''–'' explicitly refers to standard input.

The meanings of the options are:

list A comma-separated list of integer field numbers (in increasing order), with optional – to indicate ranges [e.g., **1,4,7**; **1–3,8**; **–5,10** (short for **1–5,10**); or **3–** (short for third through last field)].

–**c***list* The *list* following –**c** (no space) specifies character positions (e.g., –**c1–72** would pass the first 72 characters of each line).

–**f***list* The *list* following –**f** is a list of fields assumed to be separated in the file by a delimiter character (see –**d**); e.g., –**f1,7** copies the first and seventh field only. Lines with no field delimiters will be passed through intact (useful for table subheadings), unless –**s** is specified.

–**d***char* The character following –**d** is the field delimiter (–**f** option only). Default is *tab*. Space or other characters with special meaning to the shell must be quoted.

–**s** Suppresses lines with no delimiter characters in case of –**f** option. Unless specified, lines with no delimiters will be passed through untouched.

Either the –**c** or –**f** option must be specified.

Use *grep*(1) to make horizontal ''cuts'' (by context) through a file, or *paste*(1) to put files together column-wise (i.e., horizontally). To reorder columns in a table, use *cut* and *paste*.

EXAMPLES

```
cut –d: –f1,5 /etc/passwd          mapping of user IDs to names
```

name=\`who am i | cut −f1 −d" "\` to set **name** to current login name.

DIAGNOSTICS

ERROR: line too long
A line can have no more than 1023 characters or fields, or there is no new-line character.

ERROR: bad list for c / f option
Missing **−c** or **−f** option or incorrectly specified *list*. No error occurs if a line has fewer fields than the *list* calls for.

ERROR: no fields The *list* is empty.

ERROR: no delimiter Missing *char* on **−d** option.

ERROR: cannot handle multiple adjacent backspaces
Adjacent backspaces cannot be processed correctly.

WARNING: cannot open <filename>
Either *filename* cannot be read or does not exist. If multiple filenames are present, processing continues.

SEE ALSO

grep(1), paste(1).

NAME

cxref – generate C program cross-reference

SYNOPSIS

cxref [options] files

DESCRIPTION

The *cxref* command analyzes a collection of C files and attempts to build a cross-reference table. *cxref* uses a special version of *cpp* to include **#define**'d information in its symbol table. It produces a listing on standard output of all symbols (auto, static, and global) in each file separately, or, with the **–c** option, in combination. Each symbol contains an asterisk (*) before the declaring reference.

In addition to the **–D**, **–I** **–nostdinc** and **–U** options [which are interpreted just as they are by *cc*(1) and *cpp*(1)], the following *options* are interpreted by *cxref*:

–c	Print a combined cross-reference of all input files.
–h	Do not print column headers.
–w<*num*>	Width option which formats output no wider than <num> (decimal) columns. This option will default to 80 if <num> is not specified or is less than 51.
–o file	Direct output to *file.*
–s	Operate silently; do not print input file names.
–t	Format listing for 80-column width.
–v	Print on standard error the arguments passed to sub-passes.
–Wf,	To increase default options to the *xpass* pass, use options beginning with **–Wf,**. If some option needs increasing, *xpass* reports exactly the option to use, which will start with **–Wf,** . The option is similar to the **–Wf** option used with *cc*(1).

FILES

LLIBDIR	usually /usr/lib
LLIBDIR/xcpp	special version of the C preprocessor.
LLIBDIR/xpass	special version of the C compiler.

SEE ALSO

cc(1), cpp(1).

DIAGNOSTICS

Error messages are unusually cryptic, but usually mean that you cannot compile these files.

NOTES

The following are automatically defined to *xpass* by *cxref*: **sgi, mips, host_mips, unix, SVR3, SYSTYPE_SYSV, _SYSTYPE_SYSV, _MIPSEB, MIPSEB, LANGUAGE_C, _LANGUAGE_C .**

BUGS

cxref considers a formal argument in a *#define* macro definition to be a declaration of that symbol. For example, a program that *#includes* **ctype.h**, will contain many declarations of the variable **c**.

cxref considers identifiers in function prototypes as variable names, whether the function prototype is part of a function declaration or a function definition.

NAME

date – print and set the date

SYNOPSIS

date [+ format]
date [mmddhhmm[[yy] | [ccyy]]]

DESCRIPTION

If no argument is given, or if the argument begins with +, the current date and time are printed. Otherwise, the current date is set (only by super-user). The first *mm* is the month number; *dd* is the day number in the month; *hh* is the hour number (24 hour system); the second *mm* is the minute number; *cc* is the century minus one and is optional; *yy* is the last 2 digits of the year number and is optional. For example:

date 10080045

sets the date to Oct 8, 12:45 AM. The current year is the default if no year is mentioned. The system operates in GMT. *date* takes care of the conversion to and from local standard and daylight time. Only the super-user may change the date.

If *timed*(2) is running on the machine, the date is set on all machines in the network which are also running *timed*(2). However, the **–n** flag means ''no network'', i.e., it does not set the date for all the machines on the network.

If the argument begins with +, the output of *date* is under the control of the user. All output fields are of fixed size (zero padded if necessary). Each Field Descriptor is preceded by % and will be replaced in the output by its corresponding value. A single % is encoded by %%. All other characters are copied to the output without change. The string is always terminated with a new-line character. If the argument contains embedded blanks it must be quoted (see the **EXAMPLE** section).

Specifications of native language translations of month and weekday names are supported. The language used depends on the value of the environment variable **LANGUAGE** (see *environ*(5)). The month and weekday names used for a language are taken from strings in the file for that language in the **/lib/cftime** directory (see *cftime*(4)).

After successfully setting the date and time, *date* will display the new date according to the format defined in the environment variable **CFTIME** (see *environ*(5)).

Field Descriptors (must be preceded by a %):

a	abbreviated weekday name

A	full weekday name
b	abbreviated month name
B	full month name
d	day of month – 01 to 31
D	date as mm/dd/yy
e	day of month – 1 to 31 (single digits are preceded by a blank)
h	abbreviated month name (alias for %b)
H	hour – 00 to 23
I	hour – 01 to 12
j	day of year – 001 to 366
m	month of year – 01 to 12
M	minute – 00 to 59
n	insert a new-line character
p	string containing ante-meridiem or post-meridiem indicator (by default, AM or PM)
r	time as *hh*:*mm*:*ss pp* where *pp* is the ante-meridiem or post-meridiem indicator (by default, AM or PM)
R	time as hh:mm
S	second – 00 to 59
t	insert a tab character
T	time as *hh:mm:ss*
U	week number of year (Sunday as the first day of the week) – 01 to 52
w	day of week – Sunday = 0
W	week number of year (Monday as the first day of the week) – 01 to 52
x	Country-specific date format
X	Country-specific time format
y	year within century – 00 to 99
Y	year as *ccyy* (4 digits)
Z	timezone name

EXAMPLE

```
date '+DATE: %m/%d/%y%nTIME: %H:%M:%S'
```

would have generated as output:

```
DATE: 08/01/76
TIME: 14:45:05
```

DIAGNOSTICS

No permission	if you are not the super-user and you try to change the date

bad conversion	if the date set is syntactically incorrect

WARNING

Should you need to change the date while the system is running multi-user, use *sysadm*(1) *datetime*.

NOTE

Administrators should note the following: if you attempt to set the current date to one of the dates that the standard and alternate time zones change (for example, the date that daylight time is starting or ending), and you attempt to set the time to a time in the interval between the end of standard time and the beginning of the alternate time (or the end of the alternate time and the beginning of standard time), the results are unpredictable.

SEE ALSO

sysadm(1).
cftime(4), environ(5) in the *System Administrator's Reference Manual.*

NAME

dbx – a source-level debugger

SYNOPSIS

dbx [**–I** directory] [**–c** file] [**–i**] [**–k**] [**–C**] [**–p** pid] [**–P** name] [**–r**] [**–e nnnn**] [**-F**] [object-file] [core-file | arguments . . .]

DESCRIPTION

dbx is a source-level debugger that allows the user to trace the execution of a given *object-file* as well as a pool of arbitrary processes. This enhanced version of *dbx* works with *cc*(1), *f77*(1), *pc*(1), *pl1*(1), *as*(1), *CC*(1), and MIPS machine code.

The *object-file* used with the debugger is produced by specifying an appropriate option (usually **–g**) to the compiler. The resulting *object-file* contains symbol table information, including the names of all source files that the compiler translated to create the *object-file*. These source files are accessible from the debugger.

If a *core-file* is specified, or if the file *core* exists in the current directory, *dbx* can be used to look at the state of the program when it faulted.

The file */usr/lib/dbx.help* is a text file explaining features of *dbx*. Read it for additional information on assembler-level debugging, expressions in *dbx*, and examples.

Running dbx

If a *.dbxinit* file resides in the current directory or in the user's home directory, the commands in it are executed when *dbx* is invoked.

The environment variable **DBXINIT** may be used to hold dbx command line options. If present, the contents of **DBXINIT** are inserted before the command line options. This is most useful for options not recognized by *edge*(1), since it provides a way to pass options to dbx even if *edge* does not recognize them. Currently, options **-e** and **-F** are recognized by *dbx* but not *edge*.

When invoked, *dbx* recognizes these command line options:

–I *directory*
: Tells *dbx* to look in the specified directory for source files. Multiple directories can be specified by using multiple –I options. *dbx* searches for source files in the current directory and in the *object-file*'s directory whether or not –I is used.

–c *file*
: Selects a command file other than *.dbxinit*.

−i Uses interactive mode. This option does not treat #s as comments in a file. It prompts for source even when it reads from a file. With this option, *dbx* also has extra formatting as if for a terminal.

−p *pid* Debug the running process with the specified process id.

−P *name* Debug the running process with the specified name (*name* as shown in *ps*(1)).

−r Runs the *object-file* immediately. Arguments may be placed on the *dbx* command line. The *.dbxinit* file (if any) is read in and the commands in it executed after the *object-file* is run.

−e nnnn Choose a larger size for the evaluation stack. Printing a very large structure one can exhaust the default 20,000 bytes of evaluation stack. With this option you can make the stack as large as desired. For example, to make the stack 100,000 bytes:

```
-e 100000
```

−F Fastpath load the symbol table of the program to be debugged. At present, this helps only programs using FORTRAN common. For large FORTRAN programs, *dbx* startup time can be excessive. With this option, the internal fields of *common* will not be loaded, which makes *dbx* start up much faster. After *dbx* has finished its startup the variables in common will be invisible to *dbx*. The **readsyms** command allows you to read in *common* symbols when and where you need them visible.

−k Debug the operating system. When debugging a running system, */dev/kmem* should be specified as the *core-file*.

−C Suppress the automatic truncation of c++ variable names. Supplying this option causes the full long names output by *cfront* to *cc* to be visible. Normally only done when one believes *dbx* has truncated a name improperly. With this option c++ code tends to be much more difficult to work with.

The Monitor

The following commands control the *dbx* monitor. Multiple commands can be specified on the same command line by separating them with a semicolon (;).

![*string*] [*integer*] [*–integer]*
: Specifies a command from the history list.

help Prints a list of *dbx* commands, using the IRIX system *more*(1) command to display the list. If the environment variable **PAGER** is set, it uses the value of **PAGER** as the command name.

history Prints the items from the history list. The history list is *$lines* lines long (20 by default).

quit Exits *dbx*.

Controlling dbx

alias Lists all existing aliases.

alias *name*
: Lists the alias string for *name*. The alias value is inserted in quotes with escape characters added to show how the alias "*string*" was typed. See the help file (*/usr/lib/dbx.help*) **EXPRESSIONS** section for additional information on strings and character escapes.

alias *name* "*string*"
: Defines a new alias. See the help file (*/usr/lib/dbx.help*) **EXPRESSIONS** section for additional information on strings and character escapes.

alias *name name2*
: Defines a new alias.

alias *name*(*arg1*,...*argN*) "*string*"
: Defines a new alias. When using the alias the actual arguments are substituted into *string*. For example:

```
alias m(a,b) "print alf.a.b"
m(mystruct,i)
#the command is: print alf.mystruct.i
```

See the help file (*/usr/lib/dbx.help*) *alias* command and **HINTS** and **EXPRESSIONS** sections for additional information and examples.

unalias *alias_name*
: Removes the alias *alias_name* from the alias table.

delete *expression1*, ...*expressionN*
: Deletes the specified item(s) from the status list.

delete all
Deletes all items from the status list.

givenfile Reports the name of the current *givenfile* (usually called an *object-file* on this page).

givenfile *object-file*
The argument *object-file* is taken as the name of the file to debug, just as if *object-file* had been given on the *dbx* command line. Existing processes are killed and *dbx* looks up the symbol table and *core-file* for this *object-file*.

corefile Reports the current *core-file* name and whether data referencing commands actually reference the *core-file* or not. They will not reference the *core-file* if it is missing or damaged or if the process is running.

corefile *core-file*
The argument *core-file* is taken as the name of a *core-file* to debug, just as if *core-file* had been given on the *dbx* command line. References to the process will use this *core-file*.

playback input [*file*]
Replays commands that were saved with the **record input** command in a text file. If no *file* is given, the value of the debugger variable *$defaultin* is used as the file name. If *$pimode* is 1, the commands read will be shown on the current output.

playback output [*file*]
Shows debugger output that was saved with the **record output** command. Works much like the command *cat*(1). If no *file* is given, the value of the debugger variable *$defaultout* is used as the file name.

record input [*file*]
Records all commands typed to *dbx*. If no *file* is given, the value of the debugger variable *$defaultin* is used as the file name.

record output [*file*]
Records *dbx* output. Records the commands causing the output if the value of the debugger variable *$rimode* is one. If no *file* is given, the value of the debugger variable *$defaultout* is used as the file name.

record Lists currently set **record** commands.

unrecord *expression1, ...expressionN*
Deletes the specified item(s) from the record list.

unrecord *all*
: Deletes all items from the record list.

sh Calls a shell from *dbx*. If the environment variable **SHELL** is set, its value is taken as the name of the shell to use. If **SHELL** is not set, *csh*(1) is used.

sh *shell command*
: Executes a shell command.

status Lists currently set **stop**, **trace**, and **when** commands.

set Lists existing debugger variables and their values.

set *variable = expression*
: Assigns a new value to an existing variable or a new variable. If the expression result is not of type ''int'' the **set** command will leave the result symbolic. To get the expression fully evaluated, cast the result to ''int'' as in **set** *variable* = **(int)***expression*.

tag *funcname*
: Consults the current *$tagfile* to find and load the file which contains *funcname*. The file named by *$tagfile* (by default *tags*) must be built by *ctags*(1). This achieves the source-line-focus part of what **func** *funcname* does plus it finds C macros (if they have arguments and the macro is in a source file in the directory-list), which **func** cannot do.

unset *variable*
: Removes the specified debugger variable and its value from the debugger variable table.

Examining Source

/regular expression
: Searches ahead in the source code for the regular expression.

?regular expression
: Searches back in the source code for the regular expression.

directory
: Lists source directories.

directory *directory1 ... directoryN*
: Adds the new directory(s) to the previous list. Can be abbreviated **dir**. See also **use**.

edit Calls an editor on the current file from *dbx*. If the environment variable **EDITOR** is set, its value is taken as the name of the editor to call. If **EDITOR** is not set, *vi*(1) is the editor called.

edit *file* Calls an editor from *dbx.*

edit *func* Calls an editor on the file in which function *func* is defined.

file Prints the current file name.

file *file* Changes the current file to the specified file.

func Prints the current activation level.

func *integerexpression*
Moves to the specified activation level. It is an error if the current activation stack does not have *integerexpression* levels.

func *procedure*
Moves to the specified *procedure* in the current stack trace. Moves the source focus to the *procedure* even if the function is not active.

list Lists *$listwindow* lines starting at the current source line.

list *expression*
Lists *$listwindow* lines starting at line *expression.*

list *expression:expression2*
Lists *expression2* lines starting at line *expression.*

list *expression,expression2*
Lists lines *expression* thru *expression2.*

use Lists source directories.

use *directory1 ... directoryN*
Substitutes the new directories for the previous list. See also **directory**.

whatis *variable*
Prints a type declaration. *variable* may be a variable or a type. C structure/union/enumeration tags must be prepended with T_.

which *variable*
Finds the variable name currently being used.

whereis *variable*
Prints all qualifications (scopes) of the specified variable name.

Controlling Programs

assign *expression1 = expression2*
Assigns the specified expression to a specified program variable. If the *incompatible types* message appears when you try to assign a value to a pointer, use casts to make the assignment work.

Example:

```
# c definition might be struct x *y;
# to set y to the null pointer:
assign *(int *)(&y) = 0
```

cont Resume execution and wait for a break or other event.

cont *signal*
Send signal *signal* to the process, resume execution, and wait for a break or other event.

cont to *procedure*
Set a breakpoint at the first line of *procedure*, resume execution, and wait for a break or other event.

cont to *line*
Set a breakpoint at the line *line*, resume execution, and wait for a break or other event.

cont *signal* **to** *line*
Set a breakpoint at the line *line*, send signal *signal* to the process, resume execution, and wait for a break or other event.

cont *signal* **to** *procedure*
Set a breakpoint at the first line of *procedure*, send signal *signal* to the process, resume execution, and wait for a break or other event.

goto *line* Go to the specified line in the current source file and begin executing there. The line must be in the current procedure.

kill Kill the active process.

kill *pid ...*
Kill the active process(es) whose process id(s) are specified.

next [*integer*]
Step over the specified number of lines. The default is one. This command does not step into procedures. Breakpoints in procedures stepped over are honored. See *$nextbreak* and *$stepintoall* for information on modifying the behavior of **next**.

rerun [*arg1 ... argN*] [*file1*][*file2*]
Rerun the program, using the same arguments that were specified to the **run** command. If new arguments are specified, **rerun** uses those arguments.

resume Resume execution of the program, and return immediately to the *dbx* command interpreter.

resume *signal*
: Resume execution of the process, sending it signal *signal*, and return immediately to the *dbx* command interpreter.

run [*arg1* ... *argN*] [*file1*] [*file2*]
: Runs the program with the specified arguments.

return Continue execution until control returns to the next procedure up the activation stack.

return [*procedure*]
: Continue execution until control returns to the named procedure.

step [*integer*]
: Step the specified number of lines. The default is one line. This command steps only into procedures which have line numbers and which were compiled with options **-g**, **-g3**, or **-g2**.

Breakpoints in procedures stepped over are honored.

The command `set $stepintoall=1` will force **step** to step into all procedures for which *dbx* can find a source file even if the procedure was not compiled with symbols. A source file without symbols usually has its code rearranged in complex ways so that when stepping one sees the line numbers jump around oddly.

If the file for a procedure has the same name as a file *dbx* can find *dbx* will step into the procedure while reporting the source file it finds. The source file found may not be the source file the procedure was compiled from; this will be very confusing.

The command `set $stepintoall=1` will force **step** to step into all procedures. The caveats about $stepintoall being 1 apply here too.

suspend Stop the current process immediately.

syscall Prints the list of system calls in 4 sections. System calls may be caught (breakpointed) at the time the call is made or when the system call is about to return. This command prints the system calls in the four sections: 1) caught at call 2) ignored at call 3) caught at return 4) ignored at return. The system calls are all listed in */usr/include/sys.s* In all **syscall** commands, case is ignored when checking system call names (so you can type entirely in lower case in all the syscall commands).

syscall catch call

syscall ignore call

syscall catch return

syscall ignore return
: The four commands above print individual sections of the 4-section list that **syscall** prints.

syscall catch call *syscall* ...
: Make the named system calls breakpoint at the entry of the system call. A particularly useful setting is `syscall catch call exit` which will breakpoint the entry to exit(). Thus if the program is about to terminate you can do a stacktrace before the termination happens to see why exit() was called.

syscall ignore call *syscall* ...
: Make The named system calls not breakpoint at the entry of the system call.

syscall catch return *syscall* ...
: Make the named system calls breakpoint at the return from the system call.

syscall ignore return *syscall* ...
: Make the named system calls not breakpoint at the return from the system call.

syscall catch call all
: Make all system calls breakpoint at the entry to the system call.

syscall ignore call all
: Make all system calls not breakpoint at the entry to the system call.

syscall catch return all
: Make all system calls breakpoint at the return from the system call.

syscall ignore return all
: Make all system calls not breakpoint at the return from the system call.

wait
: Wait for the process to stop for an event (breakpoint, etc). This is useful after a **resume** command.

Execution tracing

catch
: Lists all signals that *dbx* catches.

catch *signal*

Adds a new signal to the catch list. A *signal* is specified as a name or as a number. For example, the interrupt signal is named INT, SIGINT, or 2. A process does not see this signal directed at it until:

1) The signal comes to *dbx* and the process is stopped, and

2) The process is continued.

If the process has not declared a signal handler for a signal, the process does not see the signal when it is continued.

ignore Lists all signals that *dbx* does not catch.

ignore *signal*

Adds a signal *signal* to the ignore list. A *signal* is specified as a name or as a number. For example, the interrupt signal is named INT, SIGINT, or 2. A process sees this signal when directed at it by itself or another process. The process responds to the signal just as if *dbx* were not present. A SIGINT signal at the keyboard is seen by *dbx* and it interrupts *dbx* as well as being sent to the process(s) being debugged. Remember that keyboard-generated signals are seen by the whole process group (meaning the IRIX process group, not the *dbx* process group).

stop, **trace**, and **when** are variations on a theme. They set up breakpoints/tracing, but do not begin or continue program execution. The remainder of this section describes these three commands. Each of them takes several optional arguments (clauses). Since the *variable* and **if** *expression* clauses appear over and over, the definition of their meaning appears only here.

The *variable* clause turns the command into a conditional command -- the condition is ''has the variable value changed?'' When *variable* is used, it may be either a variable or an expression.

If a variable is given, that variable is inspected at ''appropriate'' points. If an expression is given, that expression is assumed to be a pointer to a 32-bit value and the value-pointed-at is inspected at the ''appropriate'' points. ''Appropriate'' means either

1) Only at specific locations in the program (if **at** *line*), or

2) Every instruction in a given function (if **in** *procedure*), or

3) Every instruction (execution will be very slow).

If *variable* has changed then the result of the condition test ``has the variable's value changed?'' is ``true.'' Otherwise the result of the test is ``false.'' The old and new values are printed.

The **if** *expression* clause turns the command into a conditional command. The *expression* is evaluated at the same points as mentioned for *variable* and evaluates to ``true'' or ``false.''

If both *variable* and **if** *expression* are used, the overall test evaluates to ``true'' only if both evaluate ``true.''

If **in** *procedure* is specified, when executing that procedure all lower-level procedures are stepped over. That is, they are executed but no tracing is done in them. So nested traces in procedures don't work.

stop stops execution of the process when the **if** clause is true (if present) and the **variable** has changed (if present).

stop *variable*
Sets up to stop execution when *variable* changes. (Execution will be very slow).

stop *procedure*
Sets up to stop execution when *procedure* is entered.

stop at Sets a breakpoint at the current source line.

stop [*variable*] **at** *line*
Sets a breakpoint at the specified source line. The breakpoint is conditional if *variable* is specified.

stop [*variable*] **at** *line* **if** *expression*
Sets a conditional breakpoint at the specified source line.

stop [*variable*] **in** *procedure*
Sets a breakpoint in the specified procedure. The breakpoint is conditional if *variable* is specified.

stop [*variable*] **in** *procedure* **if** *expression*
Sets a conditional breakpoint in the specified procedure.

stop if *expression*
Sets up program execution tracing, testing the expression at each program line, and stopping when it is true. (Execution will be very slow).

trace prints information about the process when the trace conditions are satisfied, but program execution continues.

trace Step through the program a line at a time without stopping. Useful with *edge*(1), as *edge* shows the source code with a green bar on the line executing as *dbx* steps through it.

trace *variable*
When *variable* changes, print its old and new values. (Execution will be very slow).

trace *procedure*
When *procedure* is entered, print its arguments and its caller's name.

trace *variable* **at** *line*
Print *variable*'s old and new values when the source line is reached.

trace *variable* **at** *line* **if** *expression*
If the expression is true when the source line is reached, and *variable* has changed value, print the old and new values.

trace *variable* **in** *procedure*
Prints *variable* when it changes in *procedure.*

trace *variable* **in** *procedure* **if** *expression*
Prints *variable* when it changes in *procedure* if *expression* is true.

when is similar to **stop** except that rather than stopping when the conditions are met, the *command-list* (*dbx* commands separated by semi-colons) is executed. If one of the commands in the list is **stop** (with no operands) then the process will stop when the *command-list* is executed.

when if *expression* { *command-list* }
Execute *command-list* on every line executed for which *expression* is true. (Execution will be very slow).

when at *line* [**if** *expression*] { *command-list* }
Execute the specified *command-list* when the conditions are met.

when in *procedure* [**if** *expression*] { *command-list* }
Execute the specified *command-list* when the conditions are met.

when *variable* [**at** *line*] [**if** *expression*] { *command-list* }
Execute the specified *command-list* when the conditions are met. (Execution will be very slow unless **at** *line* is specified).

when *variable* [**in** *procedure*] [**if** *expression*] { *command-list* }
: Execute the specified *command-list* when the conditions are met. (Execution will be very slow.).

Examining Program State

ccall *func(arg1,arg2,...,argn)*
: Call a function with the given arguments. Regardless of the language the function was written in, the call is interpreted as if it were written in C. So normal C calling conventions are used. The **ccall** command is particularly suited to procedures or functions that do not return a value.

One can call functions that do return a value as normal expressions. For example, to call the function f which returns an integer (taking integer, double, and string arguments) and shift the results by two bits, type

```
    print f(1,3.0,"a value") << 2
```

If there is a breakpoint in a function called interactively the value returned by the function is lost and any computation (say in in an expression) following the function call is ignored. One can debug a function by setting breakpoints and calling it interactively.

Structure arguments to and structure returns from a function are not supported.

dump Prints variable information about the current procedure.

dump *procedure*
: Prints variable information about *procedure* (*procedure* must be active).

dump . Prints variable information for all procedures currently active.

down Moves down one activation level in the stack.

down *expression*
: Moves down the specified number of activation levels in the stack.

up Moves up one activation level on the stack.

up *n* Moves up the specified number of activation levels on the stack.

print *expression1,...expressionN*
: Prints the value(s) of the specified expression(s).

printf *"string", expression1,...expressionN*
Prints the value(s) of the specified expression(s) using C language string formatting.

printregs
Prints all register values.

readsyms
Reads in common symbols for the source file the current-function was defined in. **readsyms** does nothing if the *-F* option was not supplied to *dbx.* **readsyms** does nothing if it was already done on the relevant source file.

readsyms *func*
Reads in common symbols for the source file the named function was defined in. This is most useful in a startup script. Using plain **readsyms** (no *func*) is usually adequate, since unless a function is the current-function the data in it is not visible. **readsyms** does nothing if the *-F* option was not supplied to *dbx.* **readsyms** does nothing if it was already done on the relevant source file.

readsyms *"file"*
Reads in common symbols for the source file specified. Using plain **readsyms** is usually adequate to load the symbols for a file you are currently looking at. The *"file"* is most useful in a startup script. **readsyms** does nothing if the *-F* option was not supplied to *dbx.* **readsyms** does nothing if it was already done on the relevant source file.

The filename must be specified as it was to the compiler in the compile step. This can be a problem. For example, if two files named "a.f" exist, where one is in directory a and another in b, and the compile is done with `cd a ; f77 -c a.f ; cd ../b ; f77 -c a.f` then **readsyms** *"a.f"* will only find one of the files. **readsyms** *"b/a.f"* will not find either one. Use one of the other forms of **readsyms** to avoid the problem.

where Does a stack trace, which shows the current activation levels. **where** is sensitive to the exact positioning of *$curpc*. (*$curpc* is a debugger variable holding a copy of the program counter register.) If a stack trace looks unreasonable it may be because:

1) The stack was trashed by some program bug or,

2) The program counter is positioned in a function prologue.

Try **step** or **stepi**. If 2) is the problem, a **step** will solve it. Multiple **stepi** commands (there is no way to say how many except by examining the disassembled function) will also solve problem 2.

Debugging at the Machine Level

stopi and **tracei** have the same meanings and options as **stop** and **trace** except that **stopi** and **tracei** step in units of machine instructions rather than source lines. See the earlier discussion of **stop**, **trace**, and **when** for details on these complex statements.

conti Continue executing. Identical to **cont**.

conti *signal*
Continue executing after sending *signal* to the process.

conti to *address*
Set a breakpoint at *address* and continue.

conti to *procedure*
Set a breakpoint in *procedure* and continue. The breakpoint is set at the first instruction in the procedure, not at the first source line.

conti *signal* **to** *address*
Set a breakpoint at *address*, send signal *signal* to the process, and continue.

conti *signal* **in** *procedure*
Set a breakpoint in *procedure*, send signal *signal* to the process, and continue. The breakpoint is set at the first instruction in the procedure, not at the first source line.

nexti [*integer*]
Steps over the specified number of machine instructions. The default is one. This command does not step into procedures.

stepi [*integer*]
Steps the specified number of machine instructions. This command steps into procedures even if no source, symbols, or line numbers are present. The default is one instruction.

stopi [*variable*] **at** *address*
Set a breakpoint at *address*.

stopi [*variable*] **at** *address* **if** *expression*
Set a conditional breakpoint at *address*. The program will break only if *expression* is true.

stopi [*variable*] **in** *procedure*
: Set a breakpoint at the first machine instruction of *procedure*.

stopi [*variable*] **in** *procedure* **if** *expression*
: Set a conditional breakpoint at the first machine instruction of *procedure*.

stopi [*variable*] **if** *expression*
: Set a tracing breakpoint that will evaluate the expression after every machine instruction and break when *expression* is true. (Execution will be very slow.)

tracei *variable* **at** *address* [**if** *expression*]
: Traces the specified variable in machine instructions.

tracei *variable* **in** *procedure* [**if** *expression*]
: Traces the specified variable in machine instructions. (Execution of *procedure* will be very slow.)

address / *count format*
: Prints the contents of the specified address or disassembles the code for the instruction at the specified address. Repeated for a total of *count* addresses in increasing address. This might be termed the "examine forward" command.

address ? *count format*
: Prints the contents of the specified address or disassembles the code for the instruction at the specified address. Repeated for a total of *count* addresses in decreasing address. This might also be termed the "examine backward" command.

address / *count* L *val mask*
: Examine *count* 32-bit words in increasing address. Print those 32-bit words which, when or'd with *mask*, equal *val*. This command therefore searches memory for specific patterns.

./ repeats the previous examine command with increasing address.

.? repeats the previous examine command with decreasing address.

Examine command formats:

d	print a 16-bit word in decimal
D	print a 32-bit word in decimal
o	print a 16-bit word in octal
O	print a 32-bit word in octal

x	print a 16-bit word in hexadecimal
X	print a 32-bit word in hexadecimal
L	like X but use with *val mask*
b	print a byte in octal
c	print a byte as a character
s	print a string of characters that ends in a null byte
f	print a single-precision real number
g	print a double-precision real number
i	print machine instructions (disassemble)

Examples:

$curpc-20/20i
: prints 20 disassembled machine instructions starting at the current pc-20.

0x400200/100L 0xee 0xff
: prints 32-bit words starting at address 0x400200 whose least significant byte is hexadecimal ee. 100 words are inspected.

Multiple process debugging

If desired, a number of processes can be traced in a single *dbx* session. Processes can be added or subtracted from the ''process pool'' by the commands described below. A single process is identified as the ''active process'' and relevant commands will be applied to this process' execution and address space.

If a process performs an *execv()* or an *execve()* system call, it will stop itself and allow *dbx* to read its new symbol information.

If a process performs a *fork()* or *sproc*() system call, the default action by *dbx* will be to set the child running. However, if the variable *$promptonfork* is 1, the user will be prompted whether the child is to be added to the process pool. If *$promptonfork* is 2, the child is added to the process pool automatically. If the child is added to the process pool, both parent and child will remain stopped until continued by the user. If the child is not added to the pool, it is set running and only the parent is left stopped.

Many commands now take a clause **pid** *pid* (where *pid* is a numeric process id or a debugger variable holding a process id) at the end to make them apply to process *pid*. Commands that do so are **active**, **addproc**, **assign**, **catch**, **cont[i]**, **delete**, **delproc**, **directory**, **down**, **dump**, **edit**, **file**, **func**, **goto**, **ignore**, **kill**, **next**, **print**, **printf**, **printregs**, **readsyms**, **resume**,

return, showproc, status, step[i], stop[i], suspend, trace[i], up, use, wait, whatis, when, where, whereis, and **which.** Using the **pid** *pid* clause means you can apply a command to any process in the process pool even though it is not the active process.

addproc *pid*
: Add the running process with the specified process id to the process pool. The process will not be stopped until it is explicitly suspended by the user.

delproc *pid*
: Delete the specified process from the process pool. The process is released from *dbx*'s control and it begins/continues execution.

active Prints the active process.

active *pid*
: Make the process the active process.

showproc
: Print the status of all processes in the pool.

showproc *pid*
: Print the status of the specified process.

showproc all
: List all processes the user is able to debug (add to the process pool).

waitall Wait for any process currently running to breakpoint or stop for any reason. This does not make the process that stops first the active process.

Debugger variables help write multiple-process scripts independent of process id:

$lastchild
: is always set to the process id of the last child *fork*ed.

$pid0 is always set to the process id of the **given** process, (called *object-file* in the Synopsis).

Process Group Debugging Facilities

The process group facility allows a group of processes to be operated on simultaneously by a single *dbx* command. This is far more convenient to use when dealing with *sproc*ed processes than issuing individual **resume, suspend,** or breakpoint setting commands.

Whenever a process *sprocs*, if the child is added to the process pool the parent and child are added to the group list as well. The group list is simply a list of processes.

If the **pgrp** clause is added to the end of an applicable command (**delete**, **next[i]**, **readsyms**, **resume**, **stop[i]**, **status**, **suspend**, **trace[i]**, or **when**), the command is applied to all the processes in the group.

If *$groupforktoo* is 1, then *fork*ed processes are added to the group automatically just as *sproc*ed processes are.

stop *(operands)* **pgrp**, **trace** *(operands)* **pgrp**, and **when** *(operands)* **pgrp** each add to the *group history*. Thus breakpoints can be added to multiple processes with a single command. This group history is a numbered list which **showpgrp** shows.

delete *int* **pgrp** deletes the history for group number *int*. Thus breakpoints can be deleted from multiple processes with a single command.

Breakpoints set on the process group are recorded both in the group and in each process. Deleting breakpoints individually (even if set via a group command) is allowed.

The following commands are usable only on the processes in the group list.

addpgrp *pid ...*
: Adds the process ids specified to the group list. Only processes in the process pool can be added to the group list.

delpgrp *pid ...*
: Deletes the process ids specified from the group list.

showpgrp
: Shows the group process list and the group breakpoint list.

waitall Wait for any process currently running to breakpoint or stop for any reason. It waits on all running processes in the process list, not just those in the group list. It does not make the process that stops first the active process.

Predefined dbx Variables

The debugger has these predefined variables:

$addrfmt Specifies the format for addresses. This can be set to any format useful with C *printf*() for addresses. See *printf*(3S).

$casesense
: If 0, symbol names are case sensitive. If 1, symbol names are case insensitive. If 2, symbol name case sensitivity depends on the case sensitivity of the language that the symbol was defined

in.

$curevent
: Shows the last event number as seen by the **status** command.

$curline The current line in the source code being executed.

$curpc The current program counter.

$cursrcline
: The current source listing line plus one.

$datacache
: Caches information from the data space so that *dbx* only has to check the data space once. It defaults to non-zero, but it can be changed to 0 with no visible loss in performance. When debugging a running operating system, set this variable to 0.

$defaultin
: The name of the file that *dbx* uses when the **record input** or the **playback input** command is executed with no argument.

$defaultout
: The name of the file that *dbx* uses when the **record output** or the **playback output** command is executed with no argument.

$groupforktoo
: If 1, *fork*ed processes added to the process pool are automatically added to the process group. If 0 (the default), only *sproc*ed processes added to the process pool are automatically added to the process group.

$hexchars
: If non-zero, output characters in hexadecimal. This affects char variables, including those in structures. It does not affect arrays of characters, which are printed with a %.*s (for C) format. and thus are length-limited strings (unless $hexstrings is set).

$hexin If non-zero, input constants are assumed to be hexadecimal.

$hexints If non-zero, changes the default output constants to hexadecimal. Hexadecimal overrides octal.

$hexstrings
: If non-zero, print strings and character arrays in hexadecimal; otherwise, use characters. If non-zero then a null byte does not terminate the print; instead the array is printed for its length or $maxstrlen bytes, whichever is less. See $hexchars also.

$hide_anonymous_blocks
: If 1 (the default) anonymous inner blocks ({ } in C) are not shown in stack traces or counted in `up` or `down` commands. If 0 these blocks are shown and counted. Do `set $hide_anonymous_blocks=0` if you have a local variable in an inner block hiding a variable with the same name in an outer block and want to see the value of the outer variable.

$hide_mp_multi
: If 1 (the default) the internal library routine _mp_multi() is hidden so parallelized code can be debugged at the source level. If 0, _mp_multi() is not treated specially. Useful when debugging the internals of _mp_multi().

$historyevent
: The current history line.

$lastchild
: Is set by *dbx* to the process id of the last child *fork*ed or *sproc*ed.

$lines The number of lines in the history list (initially 20).

$listwindow
: Specifies how many lines the list command lists.

$main Specifies the name of the procedure that *dbx* will start with. This can be set to any procedure. It is not useful to set it, since it is used before any commands are read. It may be **"main"** or, depending on the language, may be some other string. It is set by the symbol-table reader.

$maxstrlen
: Specifies the maximum characters of a string that *dbx* will print for pointers to strings and for arrays of characters. See $hex-strings and $hexchars. Defaults to 128.

$mp_program
: If 0 (the default) *sproc*(2) is treated like *fork*(2). If 1, *sproc* is treated specially. The children are allowed to run (they will block on multi-processor synchronization code emitted by the mp compiler). Set this to 1 only if the code being run is multi-processor code. Set at 1 the code is easier to work with. See **MULTIPLE PROCESS DEBUGGING** in */usr/lib/dbx.help*.

$naptime
: `trace, step n`, and `next n` will delay $naptime 100ths of a second after every instruction. See *sginap*(2).

$nextbreak

If 0 (the default), $stepintoall controls whether `next` will behave as if $nextbreak were 1 or 2. If 1, a `next` command will single-step thru calls and will get back to exactly the next statement. If 1, **next, trace,** and **stop** commands testing variables may cause a break with a*not active* error - this is a bug which will be fixed later. If 2, a `next` command will execute calls at full speed Tracing and breakpointing commands are noticed regardless of the value of *$nextbreak*. With $nextbreak as 2 if the function calls itself (even indirectly) the `next` may stop at a later return from the function, not precisely the point of call (a stack trace may be deeper than you expect).

$octin If non-zero, input constants are assumed to be octal. Hexadecimal overrides octal.

$octints If non-zero, changes the default output constants to octal. Hexadecimal overrides octal.

$page Specifies whether to page when *dbx* output would scroll information off the current screen. A non-zero value turns on paging; a 0 turns it off.

$pagewindow

Specifies how many lines print when information runs longer than one screen. This can be changed to match the number of lines on any terminal. If set to 0, 1 is used.

$pid Current process id for operating system debugging (–k).

$pid0 Set by *dbx* to the process id of the *given* process (called *object-file* in the Synopsis).

$pimode If 1, prints input when used with the **playback input** command. If 0, **playback input** shows only the output of the commands read in.

$printdata

Used when disassembling (e.g., *$curpc*/20i). If 1, print register contents alongside disassembled instructions. If 0, just print disassembled instructions.

$printwhilestep

If 1, acts as if **step[i]** *n* was **step[i]** n times. With *edge*, the green bar steps through as program executes. If 0, (default) **step[i]** *n* steps *n* times then displays (and notifies *edge*).

$printwide
: If 1, prints output compactly (wide). If 0 (the default), arrays are printed one element per line.

$prompt The prompt for *dbx*.

$promptonfork
: If 0 (the default), *fork*ed/*sproc*ed processes are not added to the process pool. If 2, *fork*ed/*sproc*ed processes are automatically added to the the process pool. If 1, you are asked (on standard output) if the child is to be added to the process pool. The reply is taken from the current input file (which may be the screen).

$regstyle If 0, use hardware names for registers when disassembling.

$repeatmode
: Defaults to 0. If 1, a new-line on an empty line repeats the last command.

$rimode If 0, source commands will be saved with the **record input** command.

$stepintoall
: Defaults to 0. If 1, the **step** command will step into procedures without symbols if *dbx* can find a source file for the procedure. If 2, the **step** command will step into all procedures. This is discussed in more detail in the **step** command.

$tagfile The name of a file of tags, as created by *ctags*(1). Defaults to *tags*. Used by the **tag** command.

Predefined dbx Aliases

The debugger has these predefined aliases:

a	assign
b	stop at
bp	stop in
c	cont
d	delete
dir	directory
e	file
f	func
g	goto

h	history
j	status
l	list
li	$curpc/10i; set $curpc=$curpc+40
n or S	next
ni or Si	nexti
p	print
pd	printf "%d\n",
pi	playback input
po	printf "%o\n",
pr	printregs
px	printf "%x\n",
q	quit
r	rerun
ri	record input
ro	record output
source	playback input
s	step
si	stepi
t	where
u	list $curline-9:10
w	list $curline-5:10
wi	$curpc-20/10i
W	list $curline-10:20

HINTS

Put the command `set $pager="vi"` in your *.dbxinit* file (if *vi*(1) is not your favorite editor, substitute your editor's name in the command). Then when you use the *help* command, the help file will be brought up in a temporary file in your editor. You can then use your favorite search commands on the help file. The help file is far too large to use conveniently with *more*(1). If you are using *edge* the help file will come up in a separate window which you can leave on the screen while you are debugging.

dbx can be used as a calculator, since constant calculations can be done without specifying or running a program. A spelling error, like typing `oxf96` instead of `0xf96`, will provoke the error message ''no active process'' since *dbx* presumes the `oxf96` is a variable name. Example session:

```
$ dbx
(dbx) px ( 0xa5a893a + 0xf96) << 2
0x296a6340
(dbx)q
```

CAVEATS

The **where** command cannot print a complete stack trace if a) the program-counter is stopped in the middle of the prolog or epilog of a function and b) the stack-pointer and the program-counter refer to different function-environments. Use **stepi** to step forward a couple of instructions, which will complete the necessary prolog/epilog and allow **where** to report the complete stack trace. The problem will not occur if you avoid machine-level debugging.

The **where** problem can also occur if you debug an already running process (with *dbx* **−p**, or *dbx* **−P**, or **addproc**) since you will intercept the program at an arbitrary point. Use **stepi** or **step** to step forward to complete the necessary prolog/epilog and allow **where** to report the complete stack trace.

A **where** problem can also occur if a function in the currently active stack trace is hand-coded assembler with the **.mask** or **.frame** pseudo ops left off. In this case *dbx* simply cannot deal with the stack correctly. Compiler-generated code will not have this problem.

SEE ALSO

edge(1)
DBX Reference Manual
IRIS-4D Series Compiler Guide
/usr/lib/dbx.help

BUGS

If *vfork*(2) is called *dbx* will hang forever.

If *sproc*(2) is called with **PR_BLOCK** *dbx* will hang forever.

In either case, use interrupt (^C or whatever your interrupt key is) to get control so you can quit *dbx*.

NOTES

The options –w, –W, and –M are reserved for use by *edge*(1) when it starts *dbx*. *dbx* is unlikely to work properly if you supply any of these options.

NAME

dc – desk calculator

SYNOPSIS

dc [file]

DESCRIPTION

dc is an arbitrary precision arithmetic package. Ordinarily it operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained. (See *bc*(1), a preprocessor for *dc* that provides infix notation and a C-like syntax that implements functions. *Bc* also provides reasonable control structures for programs.) The overall structure of *dc* is a stacking (reverse Polish) calculator. If an argument is given, input is taken from that file until its end, then from the standard input. The following constructions are recognized:

number
: The value of the number is pushed on the stack. A number is an unbroken string of the digits 0–9. It may be preceded by an underscore (_) to input a negative number. Numbers may contain decimal points.

+ – / * % ^
: The top two values on the stack are added (+), subtracted (–), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. Any fractional part of an exponent is ignored.

s*x* The top of the stack is popped and stored into a register named *x*, where *x* may be any character. If the **s** is capitalized, *x* is treated as a stack and the value is pushed on it.

l*x* The value in register *x* is pushed on the stack. The register *x* is not altered. All registers start with zero value. If the **l** is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.

d The top value on the stack is duplicated.

p The top value on the stack is printed. The top value remains unchanged.

P Interprets the top of the stack as an ASCII string, removes it, and prints it.

f All values on the stack are printed.

q Exits the program. If executing a string, the recursion level is popped by two.

Q Exits the program. The top value on the stack is popped and the string execution level is popped by that value.

x Treats the top element of the stack as a character string and executes it as a string of *dc* commands.

X Replaces the number on the top of the stack with its scale factor.

[...] Puts the bracketed ASCII string onto the top of the stack.

<*x* >*x* =*x*
The top two elements of the stack are popped and compared. Register *x* is evaluated if they obey the stated relation.

v Replaces the top element on the stack by its square root. Any existing fractional part of the argument is taken into account, but otherwise the scale factor is ignored.

! Interprets the rest of the line as a UNIX system command.

c All values on the stack are popped.

i The top value on the stack is popped and used as the number radix for further input. **I** Pushes the input base on the top of the stack.

o The top value on the stack is popped and used as the number radix for further output.

O Pushes the output base on the top of the stack.

k The top of the stack is popped, and that value is used as a non-negative scale factor: the appropriate number of places are printed on output, and maintained during multiplication, division, and exponentiation. The interaction of scale factor, input base, and output base will be reasonable if all are changed together.

z The stack level is pushed onto the stack.

Z Replaces the number on the top of the stack with its length.

? A line of input is taken from the input source (usually the terminal) and executed.

; : are used by *bc*(1) for array operations.

EXAMPLE

This example prints the first ten values of n!:

```
[la1+dsa*pla10>y]sy
0sa1
lyx
```

SEE ALSO

bc(1).

DIAGNOSTICS

x is unimplemented
: where *x* is an octal number.

stack empty
: for not enough elements on the stack to do what was asked.

Out of space
: when the free list is exhausted (too many digits).

Out of headers
: for too many numbers being kept around.

Out of pushdown
: for too many items on the stack.

Nesting Depth
: for too many levels of nested execution.

NAME

delta – make a delta (change) to an SCCS file

SYNOPSIS

delta [–**r**SID] [–**s**] [–**n**] [–**g**list] [–**m**[mrlist]] [–**y**[comment]] [–**p**] files

DESCRIPTION

delta is used to permanently introduce into the named SCCS file changes that were made to the file retrieved by *get*(1) (called the *g-file*, or generated file).

delta makes a delta to each named SCCS file. If a directory is named, *delta* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with s.) and unreadable files are silently ignored. If a name of – is given, the standard input is read (see *WARNINGS*); each line of the standard input is taken to be the name of an SCCS file to be processed.

delta may issue prompts on the standard output depending upon certain keyletters specified and flags [see *admin*(1)] that may be present in the SCCS file (see –**m** and –**y** keyletters below).

Keyletter arguments apply independently to each named file.

–**r***SID*	Uniquely identifies which delta is to be made to the SCCS file. The use of this keyletter is necessary only if two or more outstanding *gets* for editing (**get** –**e**) on the same SCCS file were done by the same person (login name). The SID value specified with the –**r** keyletter can be either the SID specified on the *get* command line or the SID to be made as reported by the *get* command [see *get*(1)]. A diagnostic results if the specified SID is ambiguous, or, if necessary and omitted on the command line.
–**s**	Suppresses the issue, on the standard output, of the created delta's SID, as well as the number of lines inserted, deleted and unchanged in the SCCS file.
–**n**	Specifies retention of the edited *g-file* (normally removed at completion of delta processing).
–**g***list*	a *list* (see *get*(1) for the definition of *list*) of deltas which are to be *ignored* when the file is accessed at the change level (SID) created by this delta.

−m*[mrlist]*
: If the SCCS file has the v flag set [see *admin*(1)] then a Modification Request (MR) number *must* be supplied as the reason for creating the new delta.

  If **−m** is not used and the standard input is a terminal, the prompt MRs? is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The MRs? prompt always precedes the **comments?** prompt (see −y keyletter).

  MRs in a list are separated by blanks and/or tab characters. An unescaped new-line character terminates the MR list.

  Note that if the v flag has a value [see *admin*(1)], it is taken to be the name of a program (or shell procedure) which will validate the correctness of the MR numbers. If a non-zero exit status is returned from the MR number validation program, *delta* terminates. (It is assumed that the MR numbers were not all valid.)

−y*[comment]*
: Arbitrary text used to describe the reason for making the delta. A null string is considered a valid *comment*.

  If −y is not specified and the standard input is a terminal, the prompt **comments?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped new-line character terminates the comment text.

−p
: Causes *delta* to print (on the standard output) the SCCS file differences before and after the delta is applied in a *diff*(1) format.

FILES

g-file	Existed before the execution of *delta*; removed after completion of *delta*.
p-file	Existed before the execution of *delta*; may exist after completion of *delta*.
q-file	Created during the execution of *delta*; removed after completion of *delta*.

x-file	Created during the execution of *delta*; renamed to SCCS file after completion of *delta*.
z-file	Created during the execution of *delta*; removed during the execution of *delta*.
d-file	Created during the execution of *delta*; removed after completion of *delta*.
/usr/bin/bdiff	Program to compute differences between the ‘‘gotten’’ file and the *g-file*.

WARNINGS

Lines beginning with an SOH ASCII character (binary 001) cannot be placed in the SCCS file unless the **SOH** is escaped. This character has special meaning to SCCS [see *sccsfile*(4) (5)] and will cause an error.

A *get* of many SCCS files, followed by a *delta* of those files, should be avoided when the *get* generates a large amount of data. Instead, multiple *get*/*delta* sequences should be used.

If the standard input (–) is specified on the *delta* command line, the **–m** (if necessary) and –y keyletters *must* also be present. Omission of these keyletters causes an error to occur.

Comments are limited to text strings of at most 512 characters.

SEE ALSO

admin(1), cdc(1), get(1), prs(1), rmdel(1), sccsfile(4).
bdiff(1), help(1) in the *User's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

deroff – remove nroff/troff, tbl, and eqn constructs

SYNOPSIS

deroff [**–i**] [**–mx**] [**–w**] [files]

DESCRIPTION

deroff reads each of the *files* in sequence and removes all *troff*(1) requests, macro calls, backslash constructs, *eqn*(1) constructs (between **.EQ** and **.EN** lines, and between delimiters), and *tbl*(1) descriptions, perhaps replacing them with white space (blanks and blank lines), and writes the remainder of the file on the standard output. *deroff* follows chains of included files (**.so** and **.nx** *troff* commands); if a file has already been included, a **.so** naming that file is ignored and a **.nx** naming that file terminates execution. If no input file is given, *deroff* reads the standard input.

The **–m** option may be followed by an **m**, **s**, or **l**. The **–mm** option causes the macros to be interpreted so that only running text is output (i.e., no text from macro lines.) The **–ml** option forces the **–mm** option and also causes deletion of lists associated with the **mm** macros.

The **–i** option causes *deroff* to ignore **.so** and **.nx** commands.

If the **–w** option is given, the output is a word list, one "word" per line, with all other characters deleted. Otherwise, the output follows the original, with the deletions mentioned above. In text, a "word" is any string that *contains* at least two letters and is composed of letters, digits, ampersands (&), and apostrophes ('); in a macro call, however, a "word" is a string that *begins* with at least two letters and contains a total of at least three letters. Delimiters are any characters other than letters, digits, apostrophes, and ampersands. Trailing apostrophes and ampersands are removed from "words."

SEE ALSO

eqn(1), nroff(1), tbl(1), troff(1) in the *DOCUMENTER'S WORKBENCH Software Release 2.0 Technical Discussion and Reference Manual.*

BUGS

deroff is not a complete *troff* interpreter, so it can be confused by subtle constructs. Most such errors result in too much rather than too little output. The **–ml** option does not handle nested lists correctly.

NAME

df – report number of free disk blocks

SYNOPSIS

df [**–b**] [**–f**] [**–i**] [**–k**] [file-system ...]

DESCRIPTION

df reports the number of total, used, and available disk blocks (one disk block equals 512 bytes) in file systems. The *file-system* argument may name a device special file containing a disk filesystem, a mounted NFS filesystem of the form *hostname:pathname*, or any file, directory, or special node in a mounted filesystem. If no *file-system* arguments are specified, *df* reports on all mounted file systems.

The **–b** flag causes *df* to report usage in 512-byte units, which is the default. The **–k** flag causes *df* to report usage in 1024-byte units.

Normally, the free block information is gleaned from the file system's superblock. The **–f** flag forces a scan of the free block list.

The **–i** flag reports the number and percentage of used inodes and the number of free inodes.

The **–q** and **–t** flags are recognized but ignored. They are provided for compatibility with previous releases.

EXAMPLES

To report usage in the root filesystem, use either of the following:

```
df /dev/root
df  /
```

Report on the file system containing the current directory:

```
df  .
```

FILES

/etc/mtab

SEE ALSO

statfs(2), fs(4), mntent(4).

BUGS

Free counts may be incorrect, with or without the **–f** flag.

If *file-system* names an NFS file in a filesystem exported with the *–nohide* option on the server (see *exportfs*(1M)), and the client mounts an ancestor of that filesystem, then *df* will report incorrect information.

NOTES

In previous IRIX releases, usage was reported in 1024-byte units.

Although the *dbg* file system (normally mounted under */debug)* consumes no actual disk space, the total and free blocks reported respectively represent the total virtual memory (real memory plus swap space) present and the amount currently free.

The –i option applied filesystems of type *nfs* reports a free inode count of 0. Future versions of NFS will support useful inode counts. For the *dbg* filesystem type, –i reports the number of active process slots in the *iuse* column, and reports the number of available slots in the *ifree* column.

NAME

dglfax – electronic fax program

SYNOPSIS

dglfax [**-r**] [**-usecs**] rhost

DESCRIPTION

Dglfax is a program for sending an electronic fax (a section of your IRIS workstation's screen) to the remote host system **rhost.** You must use either an equivalent account on the remote host or an account that has no password. Remote hostnames may also take the form ''rname@rhost'' to use **rname** rather than the current username on the remote host.

When you (the sender) run *dglfax,* 4Sight will prompt you for a window. Using the mouse, enter in the area of the screen you wish to send. Once specified, the window will be moved off the screen, a window opened on the destination workstation, and the pixels read and sent to the destination.

The receiver of the fax will see a window appear on his screen in the same location as your original window. In addition, the keyboard bell is rung to alert him that he has received a fax. The header of the window identifies the sender of the fax in username@hostname format. The image in the window is a snapshot of the sender's screen. If the window is moved and redrawn, the same image will appear even if the sender's screen has changed. The image can be manually updated (pulled) from the sender's screen using the popup menu. Whenever the image is being read from the sender's screen the word "from" in the window header changes to "<---".

One problem with *dglfax* is that it draws the entire picture in either RGB or colorindex mode. If different parts of the picture are in different modes, then it cannot draw it correctly. The popup menu allows the receiver to toggle between colorindex mode and RGBmode. The default startup mode is colorindex mode.

POPUPS

Pressing the right mouse button in the window brings up the popup menu. The menu has three entries: **pull, colorindex/RGB,** and **quit.** Selecting **quit** exits the program and the window disappears. Selecting **pull** instructs the sending workstation to re-read the screen and send the new image. Note that simply redraws of the window do not re-read the sender's screen and will not result in a newer picture. The second popup menu entry is either **colorindex** if the picture is in RGB mode, or **RGB** if the picture is in colorindex mode. Selecting the second entry toggles the color mode and the menu entry and also pulls the image from the sending workstation.

Dglfax uses the Distributed Graphics Library for sending the pixels over the network. On the destination workstation, a dgld server process is run and receives the Graphics Library commands from the sender.

OPTIONS

The following options are supported:

-r	This option sets the default startup mode to RGB.
-u secs	This option causes an update every 'secs' seconds. Each update re-reads the sender's screen and then draws the image in the destination window.

FILES

/usr/sbin/dglfax

NAME

diff – differential file and directory comparator

SYNOPSIS

diff [**–l**] [**–r**] [**–s**] [**–cefhn**] [**–biwt**] dir1 dir2
diff [**–cefhn**] [**–biwt**] file1 file2
diff [**–D***string*] [**–biw**] file1 file2

DESCRIPTION

If both arguments are directories, *diff* sorts the contents of the directories by name, and then runs the regular file *diff* algorithm (described below) on text files which are different. Binary files which differ, common subdirectories, and files which appear in only one directory are listed. Options when comparing directories are:

–l long output format; each text file *diff* is piped through *pr*(1) to paginate it, other differences are remembered and summarized after all text file differences are reported.

–r causes application of *diff* recursively to common subdirectories encountered.

–s causes *diff* to report files which are the same, which are otherwise not mentioned.

–Sname
starts a directory *diff* in the middle beginning with file *name.*

When run on regular files, and when comparing text files which differ during directory comparison, *diff* tells what lines must be changed in the files to bring them into agreement. Except in rare circumstances, *diff* finds a smallest sufficient set of file differences. If neither *file1* nor *file2* is a directory, then either may be given as '–', in which case the standard input is used. If *file1* is a directory, then a file in that directory whose file-name is the same as the file-name of *file2* is used (and vice versa).

There are several options for output format; the default output format contains lines of these forms:

n1 a *n3*,*n4*
n1,*n2* d *n3*
n1,*n2* c *n3*,*n4*

These lines resemble *ed* commands to convert *file1* into *file2*. The numbers after the letters pertain to *file2*. In fact, by exchanging 'a' for 'd' and reading backward one may ascertain equally how to convert *file2* into *file1*. As in *ed,* identical pairs where *n1* = *n2* or *n3* = *n4* are abbreviated as a single number.

Following each of these lines come all the lines that are affected in the first file flagged by '<', then all the lines that are affected in the second file flagged by '>'.

Except for **–b**, **-w**, **-i** or **-t** which may be given with any of the others, the following options are mutually exclusive:

–e produces a script of *a, c* and *d* commands for the editor *ed,* which will recreate *file2* from *file1*. In connection with **–e**, the following shell program may help maintain multiple versions of a file. Only an ancestral file ($1) and a chain of version-to-version *ed* scripts ($2,$3,...) made by *diff* need be on hand. A 'latest version' appears on the standard output.

```
(shift; cat $*; echo ´1,$p`) | ed – $1
```

Extra commands are added to the output when comparing directories with **–e**, so that the result is a *sh*(1) script for converting text files which are common to the two directories from their state in *dir1* to their state in *dir2*.

–f produces a script similar to that of **–e**, not useful with *ed,* and in the opposite order.

–n produces a script similar to that of **–e**, but in the opposite order and with a count of changed lines on each insert or delete command. This is the form used by the RCS commands for storing a revision change.

–c produces a diff with lines of context. The default is to present 3 lines of context and may be changed, e.g to 10, by **–c10**. With **–c** the output format is modified slightly: the output beginning with identification of the files involved and their creation dates and then each change is separated by a line with a dozen *'s. The lines removed from *file1* are marked with '– '; those added to *file2* are marked '+ '. Lines which are changed from one file to the other are marked in both files with with '! '.

Changes which lie within <context> lines of each other are grouped together on output. (This is a change from the previous "diff -c" but the resulting output is usually much easier to interpret.)

–h does a fast, half-hearted job. It works only when changed stretches are short and well separated, but does work on files of unlimited length.

–Dstring causes *diff* to create a merged version of *file1* and *file2* on the standard output, with C preprocessor controls included so that a compilation of the result without defining *string* is equivalent to compiling *file1*, while defining *string* will yield *file2*.

–b causes trailing blanks (spaces and tabs) to be ignored, and other strings of blanks to compare equal.

–w is similar to **–b** but causes whitespace (blanks and tabs) to be totally ignored. E.g., ``if ( a == b )'' will compare equal to ``if(a==b)''.

–i ignores the case of letters. E.g., ``A'' will compare equal to ``a''.

–t will expand tabs in output lines. Normal or **–c** output adds character(s) to the front of each line which may screw up the indentation of the original source lines and make the output listing difficult to interpret. This option will preserve the original source's indentation.

FILES

/bin/diff	executable for *diff.*
/usr/lib/diffh	executable used for the **-h** option.
/tmp/d?????	working files.
/bin/pr	executed by the **-l** option.

SEE ALSO

bdiff(1), cc(1), cmp(1), comm(1), diff3(1), ed(1)

DIAGNOSTICS

Exit status is 0 for no differences, 1 for some differences, 2 for trouble.

BUGS

Editing scripts produced under the **–e** or **–f** option are naive about creating lines consisting of a single '**.**'.

When comparing directories with the **–b**, **-w** or **-i** options specified, *diff* first compares the files a la *cmp*(1), and then decides to run the *diff* algorithm if they are not equal. This may cause a small amount of spurious output if the files then turn out to be identical because the only differences are insignificant blank string or case differences.

NAME

diff3 – 3-way differential file comparison

SYNOPSIS

diff3 [**–exEX3**] file1 file2 file3

DESCRIPTION

Diff3 compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

====	all three files differ
====1	*file1* is different
====2	*file2* is different
====3	*file3* is different

The type of change suffered in converting a given range of a given file to some other is indicated in one of these ways:

f : *n1* **a**	Text is to be appended after line number *n1* in file *f*, where *f* = 1, 2, or 3.
f : *n1* , *n2* **c**	Text is to be changed in the range line *n1* to line *n2*. If *n1* = *n2*, the range may be abbreviated to *n1*.

The original contents of the range follows immediately after a **c** indication. When the contents of two files are identical, the contents of the lower-numbered file is suppressed.

Under the **–e** option, *diff3* publishes a script for the editor *ed* that will incorporate into *file1* all changes between *file2* and *file3*, *i.e.* the changes that normally would be flagged ==== and ====3. Option **–x** (**–3**) produces a script to incorporate only changes flagged ==== (====3). The following command will apply the resulting script to 'file1'.

```
(cat script; echo '1,$p') | ed – file1
```

The **–E** and **–X** are similar to **–e** and **–x**, respectively, but treat overlapping changes (i.e., changes that would be flagged with ==== in the normal listing) differently. The overlapping lines from both files will be inserted by the edit script, bracketed by "<<<<<<" and ">>>>>>" lines.

For example, suppose lines 7-8 are changed in both file1 and file2. Applying the edit script generated by the command

"diff3 -E file1 file2 file3"

to file1 results in the file:

```
lines 1-6
of file1
<<<<<<< file1
```

```
lines 7-8
of file1
=======
lines 7-8
of file3
>>>>>>> file3
rest of file1
```

The −E option is used by RCS *merge*(1) to insure that overlapping changes in the merged files are preserved and brought to someone's attention.

FILES

/tmp/d3*
/usr/lib/diff3prog

SEE ALSO

diff(1).

BUGS

Text lines that consist of a single . will defeat −e.

DIAGNOSTICS

"Too many changes" means the number of differences between either pairs of files exceeds 2000.

If execution is successful, *diff3*'s exit status is the number of overlaps found.

NAME

dircmp – directory comparison

SYNOPSIS

dircmp [**–d**] [**–s**] [**–w***n*] dir1 dir2

DESCRIPTION

dircmp examines *dir1* and *dir2* and generates various tabulated information about the contents of the directories. Listings of files that are unique to each directory are generated for all the options. If no option is entered, a list is output indicating whether the file names common to both directories have the same contents.

–d Compare the contents of files with the same name in both directories and output a list telling what must be changed in the two files to bring them into agreement. The list format is described in *diff*(1).

–s Suppress messages about identical files.

–w*n* Change the width of the output line to *n* characters. The default width is 72.

SEE ALSO

cmp(1), diff(1).

NAME

dirview – graphical interface to file system

SYNOPSIS

dirview pathnames
dirview **–o** sourcename targetname

DESCRIPTION

dirview is a tool which allows quick access to a directory view not currently open on the WorkSpace. When invoked with a directory path name, it will request an existing WorkSpace process to open a directory view of that directory. If no WorkSpace process exists for that user, dirview will start one.

dirview accepts the following options.

pathname [pathname...]
If *dirview* is run with one or more directory pathnames as arguments, a view will be opened for each of those directories.

–o sourcename targetname
If a view is open for *sourcename*, replace it with a view of *targetname*. *sourcename* and *targetname* can be either full or relative path names.

SEE ALSO

workspace(1G)

Programming the IRIS WorkSpace

NAME

dis – disassemble an object file

SYNOPSIS

dis [-h] [-d] [-S] [-Idirectory] [-s secname] [-p procedure] [file ...]

DESCRIPTION

dis disassembles object files into machine instructions. A *file* can be an object, an archive library, or an executable. The options are as follows:

–h substitute the general register names for the software register names in the output.

–d remove the leading source line and hexadecimal value of disassembly, leaving only the instructions.

–S display source code intermixed with the assembly code.

–I*directory*
use the *directory* to help find the source code.

–s *secname*
disassemble the object file section *secname*. This defaults to *.text*. Use this option to disassemble other sections containing instructions (e.g., *.init*).

–p *procedure*
disassemble only the specified procedure from the object file.

BUGS

Only one –I is allowed.

NAME

disks – interactive local and network disk mounts tool

SYNOPSIS

disks

DESCRIPTION

The *disks* command is an interactive tool used for managing and perusing local and networked (NFS) mounts. *disks* displays all local and specified remote disk partitions that can or are already mounted.

FILES

/usr/lib/vadmin
/usr/lib/vadmin/filetypes

SEE ALSO

NAME

dumpfont – dump font out in some other format

SYNOPSIS

dumpfont [**–a** | **–b** | **–v** | **–vf** | **–fm**] [**–c** *comment*] [**–d** *dirname*] [**–f** *n*]
[**–n** *fontname*] [**–S**] [**–s** *n*] [**–t**] [**–tv**] [**–ta**] *filenames*

DESCRIPTION

dumpfont reads in the set of named font files and dumps them out again according to the specified options, effectively converting the files from one font format to another. *dumpfont* is typically used to generate fonts for use with the NeWS window system.

There are five types of font files that *dumpfont* can read: Sun standard vfont format, Adobe ASCII bitmap format, Adobe ASCII metric format, NeWS font format, and CMU (Andrew) format. The format of the input font is determined automatically by inspecting the file. It can write fonts out in one of four formats: Adobe ASCII, NeWS, vfont, and Silicon Graphics Font Manager format. The default output format is NeWS.

OPTIONS

–a Selects Adobe ASCII output format. This is the format that you should use when transporting fonts from one machine architecture to another. The output file extension is ".afont".

–b Selects NeWS output format (the default). The output file extension is ".font". If the input file is an Adobe ASCII metrics file, the extension will be ".metrics".

–v Selects vfont output format. The output file extension is ".vfont".

–vf Selects vfont output format. The output file extension is ".vfont". Forces the characters to be fixed width.

–fm Selects SGI Font Manager output format. The output file extension is ".fm" or ".fw", for font data or font width data, respectively. *dumpfont* only writes this format, it does not read it.

−c *comment*	Sets the *comment* field of the font. The Adobe ASCII and NeWS font formats support an internal comment that accompanies the font. This is usually used to contain copyright or history information. It is normally propagated automatically.
−d *dirname*	Specifies the directory into which the font files will be written. If the FONTDIR environment variable is set, it is used as the default value. Otherwise, if the NEWSHOME environment variable is set, *$NEWSHOME/fonts* is used as the default value. Otherwise "." is used.
−f *n*	Sets the maximum length of an output filename (excluding extension) to *n* . When writing NeWS format files, NeWS normally constructs the output filename from the name of the font and its scaling factors. Some systems cannot cope with long file names, so this option can be used to heuristically squeeze the name.
−n *name*	Forces the output font name to be *name* . It is important not to confuse the name of the font with the name of the file that contains it. Some font formats (Adobe ASCII and NeWS) contain the name of the font internally. So given a 10-point Times-Roman font, its font name will be "Times-Roman", but its file name might be *TimRom10.font.*
−S	Attempts to determine the size information of fonts by inspecting the bitmaps and applying some heuristics. This is useful when reading vfonts (particularly those intended for printers like the Versatec) that are missing or have incorrect size information.
−s *n*	Sets the point size of the font to *n* . Overrides any internal size specification
−t	Prints a short description of the fonts on standard output; a reformatted font file is not dumped.

–tv	Prints a move verbose description of the fonts on standard output; a reformatted font file is not dumped.
–ta	Prints a long description of the fonts on standard output; a reformatted font file is not dumped. You'll get every scrap of information.

SEE ALSO

bldfamily(1), vfont(5).
4Sight User's Guide, Section 2, "Programming in NeWS."

DIAGNOSTICS

Bad flag: -C	Unknown command like option
Couldn't write ...	Error writing font file
f: not a valid font or not there.	The input file does not have a valid format, or the file cannot be found.

E – F

NAME

ecc – dump memory ecc log

SYNOPSIS

ecc [-c]

DESCRIPTION

ecc dumps the memory ecc log. This log is produced by the memory subsystem each time a memory error occurs. Types of errors are single data or check bit errors (which are automatically corrected) and double bit errors (which are uncorrectable). Some uncorrectable memory errors will cause programs to be killed or the system to crash. Correctable errors, while ok, should be watched. Too many correctable errors in a given memory bank may be an early warning signal as to future more serious problems. The *-c* option will cause the log to be cleared.

This command only functions on systems which have error correcting memory.

Note that on system crash and subsequent reset the log contents are not cleared, and may be read via the prom monitor. Upon system boot, the log is cleared.

The various memory configurations cannot be determined by software, so *ecc* prints out the SIM locations for all possible configurations. It is up to service personnel to determine which is appropriate for a given system.

NAME

echo – echo arguments

SYNOPSIS

echo [arg] ...

DESCRIPTION

echo writes its arguments separated by blanks and terminated by a new-line on the standard output. It also understands C-like escape conventions; beware of conflicts with the shell's use of \:

\b	backspace
\c	print line without new-line
\f	form-feed
\n	new-line
\r	carriage return
\t	tab
\v	vertical tab
\\	backslash
\0*n*	where *n* is the 8-bit character whose ASCII code is the 1-, 2- or 3-digit octal number representing that character.

echo is useful for producing diagnostics in command files and for sending known data into a pipe.

SEE ALSO

sh(1).

CAVEATS

When representing an 8-bit character by using the escape convention **\0***n*, the *n* must always be preceded by the digit zero (0).

For example, typing: **echo 'WARNING:\07'** will print the phrase **WARNING:** and sound the ''bell'' on your terminal. The use of single (or double) quotes (or two backslashes) is required to protect the ''\'' that precedes the ''07''.

For the octal equivalents of each character, see ascii(5), in the *Programmer's Reference Manual.*

NAME

ed, red – text editor

SYNOPSIS

ed [–s] [**–p** string] [–x] [–C] [file]

red [–s] [**–p** string] [–x] [–C] [file]

DESCRIPTION

ed is the standard text editor. If the *file* argument is given, *ed* simulates an *e* command (see below) on the named file; that is to say, the file is read into *ed*'s buffer so that it can be edited.

–s Suppresses the printing of character counts by *e*, *r*, and *w* commands, of diagnostics from *e* and *q* commands, and of the **!** prompt after a !*shell command*.

–p Allows the user to specify a prompt string.

–x Encryption option; when used, *ed* simulates an **X** command and prompts the user for a key. This key is used to encrypt and decrypt text using the algorithm of *crypt*(1). The **X** command makes an educated guess to determine whether text read in is encrypted or not. The temporary buffer file is encrypted also, using a transformed version of the key typed in for the –x option. See *crypt*(1). Also, see the **WARNINGS** section at the end of this manual page.

–C Encryption option; the same as the –x option, except that *ed* simulates a **C** command. The **C** command is like the **X** command, except that all text read in is assumed to have been encrypted.

ed operates on a copy of the file it is editing; changes made to the copy have no effect on the file until a *w* (write) command is given. The copy of the text being edited resides in a temporary file called the *buffer*. There is only one buffer.

red is a restricted version of *ed*. It will only allow editing of files in the current directory. It prohibits executing shell commands via !*shell command*. Attempts to bypass these restrictions result in an error message (*restricted shell*).

Both *ed* and *red* support the *fspec*(4) formatting capability. After including a format specification as the first line of *file* and invoking *ed* with your terminal in **stty –tabs** or **stty tab3** mode (see *stty*(1)), the specified tab stops will automatically be used when scanning *file*. For example, if the first line of a file contained:

```
        <:t5,10,15 s72:>
```

tab stops would be set at columns 5, 10, and 15, and a maximum line length of 72 would be imposed. NOTE: when you are entering text into the file, this format is not in effect; instead, because of being in **stty −tabs** or **stty tab3** mode, tabs are expanded to every eighth column.

Commands to *ed* have a simple and regular structure: zero, one, or two *addresses* followed by a single-character *command*, possibly followed by parameters to that command. These addresses specify one or more lines in the buffer. Every command that requires addresses has default addresses, so that the addresses can very often be omitted.

In general, only one command may appear on a line. Certain commands allow the input of text. This text is placed in the appropriate place in the buffer. While *ed* is accepting text, it is said to be in *input mode*. In this mode, *no* commands are recognized; all input is merely collected. Leave input mode by typing a period (.) at the beginning of a line, followed immediately by a carriage return.

ed supports a limited form of *regular expression* notation; regular expressions are used in addresses to specify lines and in some commands (e.g., *s*) to specify portions of a line that are to be substituted. A regular expression (RE) specifies a set of character strings. A member of this set of strings is said to be *matched* by the RE. The REs allowed by *ed* are constructed as follows:

The following *one-character REs* match a *single* character:

1.1 An ordinary character (*not* one of those discussed in 1.2 below) is a one-character RE that matches itself.

1.2 A backslash (\) followed by any special character is a one-character RE that matches the special character itself. The special characters are:

a. ., *, [, and \ (period, asterisk, left square bracket, and backslash, respectively), which are always special, *except* when they appear within square brackets ([ ]; see 1.4 below).

b. ^ (caret or circumflex), which is special at the *beginning* of an *entire* RE (see 3.1 and 3.2 below), or when it immediately follows the left of a pair of square brackets ([ ]) (see 1.4 below).

c. $ (dollar sign), which is special at the *end* of an entire RE (see 3.2 below).

d. The character used to bound (i.e., delimit) an entire RE, which is special for that RE (for example, see how slash (/) is used in the *g* command, below.)

1.3 A period (**.**) is a one-character RE that matches any character except new-line.

1.4 A non-empty string of characters enclosed in square brackets (**[ ]**) is a one-character RE that matches *any one* character in that string. If, however, the first character of the string is a circumflex (**^**), the one-character RE matches any character *except* new-line and the remaining characters in the string. The **^** has this special meaning *only* if it occurs first in the string. The minus (**–**) may be used to indicate a range of consecutive ASCII characters; for example, **[0–9]** is equivalent to **[0123456789]**. The **–** loses this special meaning if it occurs first (after an initial **^**, if any) or last in the string. The right square bracket (**]**) does not terminate such a string when it is the first character within it (after an initial **^**, if any); e.g., **[ ]a–f]** matches either a right square bracket (**]**) or one of the letters **a** through **f** inclusive. The four characters listed in 1.2.a above stand for themselves within such a string of characters.

The following rules may be used to construct *RE*s from one-character REs:

2.1 A one-character RE is a RE that matches whatever the one-character RE matches.

2.2 A one-character RE followed by an asterisk (*****) is a RE that matches *zero* or more occurrences of the one-character RE. If there is any choice, the longest leftmost string that permits a match is chosen.

2.3 A one-character RE followed by **\{***m***\}**, **\{***m,***\}**, or **\{***m,n***\}** is a RE that matches a *range* of occurrences of the one-character RE. The values of *m* and *n* must be non-negative integers less than 256; **\{***m***\}** matches *exactly m* occurrences; **\{***m,***\}** matches *at least m* occurrences; **\{***m,n***\}** matches *any number* of occurrences *between m* and *n* inclusive. Whenever a choice exists, the RE matches as many occurrences as possible.

2.4 The concatenation of REs is a RE that matches the concatenation of the strings matched by each component of the RE.

2.5 A RE enclosed between the character sequences **\(** and **\)** is a RE that matches whatever the unadorned RE matches.

2.6 The expression *****n* matches the same string of characters as was matched by an expression enclosed between **\(** and **\)** *earlier* in the same RE. Here *n* is a digit; the sub-expression specified is that beginning with the *n*-th occurrence of **\(** counting from the left. For example, the expression **^\(.*\)\1$** matches a line consisting of two repeated appearances of the same string.

Finally, an *entire RE* may be constrained to match only an initial segment or final segment of a line (or both).

3.1 A circumflex (^) at the beginning of an entire RE constrains that RE to match an *initial* segment of a line.

3.2 A dollar sign ($) at the end of an entire RE constrains that RE to match a *final* segment of a line.

The construction ^*entire RE*$ constrains the entire RE to match the entire line.

The null RE (e.g., //) is equivalent to the last RE encountered. See also the last paragraph before **FILES** below.

To understand addressing in *ed* it is necessary to know that at any time there is a *current line*. Generally speaking, the current line is the last line affected by a command; the exact effect on the current line is discussed under the description of each command. *Addresses* are constructed as follows:

1. The character **.** addresses the current line.
2. The character **$** addresses the last line of the buffer.
3. A decimal number *n* addresses the *n*-th line of the buffer.
4. *'x* addresses the line marked with the mark name character *x*, which must be an ASCII lower-case letter (a-z). Lines are marked with the *k* command described below.
5. A RE enclosed by slashes (/) addresses the first line found by searching *forward* from the line *following* the current line toward the end of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the beginning of the buffer and continues up to and including the current line, so that the entire buffer is searched. See also the last paragraph before **FILES** below.
6. A RE enclosed in question marks (?) addresses the first line found by searching *backward* from the line *preceding* the current line toward the beginning of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the end of the buffer and continues up to and including the current line. See also the last paragraph before **FILES** below.
7. An address followed by a plus sign (+) or a minus sign (–) followed by a decimal number specifies that address plus (respectively minus) the indicated number of lines. The plus sign may be omitted.

8. If an address begins with + or –, the addition or subtraction is taken with respect to the current line; e.g, **–5** is understood to mean **.–5**.
9. If an address ends with + or –, then 1 is added to or subtracted from the address, respectively. As a consequence of this rule and of Rule 8, immediately above, the address – refers to the line preceding the current line. (To maintain compatibility with earlier versions of the editor, the character ^ in addresses is entirely equivalent to –.) Moreover, trailing + and – characters have a cumulative effect, so –– refers to the current line less 2.
10. For convenience, a comma (,) stands for the address pair **1,$**, while a semicolon (;) stands for the pair **.,$**.

Commands may require zero, one, or two addresses. Commands that require no addresses regard the presence of an address as an error. Commands that accept one or two addresses assume default addresses when an insufficient number of addresses is given; if more addresses are given than such a command requires, the last one(s) are used.

Typically, addresses are separated from each other by a comma (,). They may also be separated by a semicolon (;). In the latter case, the current line (.) is set to the first address, and only then is the second address calculated. This feature can be used to determine the starting line for forward and backward searches (see Rules 5 and 6, above). The second address of any two-address sequence must correspond to a line that follows, in the buffer, the line corresponding to the first address.

In the following list of *ed* commands, the default addresses are shown in parentheses. The parentheses are *not* part of the address; they show that the given addresses are the default.

It is generally illegal for more than one command to appear on a line. However, any command (except *e*, *f*, *r*, or *w*) may be suffixed by **l**, **n**, or **p** in which case the current line is either listed, numbered or printed, respectively, as discussed below under the *l*, *n*, and *p* commands.

(.)a
<text>
.

The *a*ppend command reads the given text and appends it after the addressed line; . is left at the last inserted line, or, if there were none, at the addressed line. Address 0 is legal for this command: it causes the ''appended'' text to be placed at the beginning of the buffer. The maximum number of characters that may be entered from a terminal is 256 per line (including the new-line character).

(.)c
<text>
.

The *c*hange command deletes the addressed lines, then accepts input text that replaces these lines; **.** is left at the last line input, or, if there were none, at the first line that was not deleted.

C

Same as the **X** command, except that *ed* assumes all text read in for the **e** and **r** commands is encrypted unless a null key is typed in.

(.,.)d

The *d*elete command deletes the addressed lines from the buffer. The line after the last line deleted becomes the current line; if the lines deleted were originally at the end of the buffer, the new last line becomes the current line.

e *file*

The *e*dit command causes the entire contents of the buffer to be deleted, and then the named file to be read in; **.** is set to the last line of the buffer. If no file name is given, the currently remembered file name, if any, is used (see the *f* command). The number of characters read is typed; *file* is remembered for possible use as a default file name in subsequent *e*, *r*, and *w* commands. If *file* is replaced by **!**, the rest of the line is taken to be a shell (*sh*(1)) command whose output is to be read. Such a shell command is *not* remembered as the current file name. See also **DIAGNOSTICS** below.

E *file*

The *E*dit command is like *e*, except that the editor does not check to see if any changes have been made to the buffer since the last *w* command.

f *file*

If *file* is given, the *f*ile-name command changes the currently remembered file name to *file*; otherwise, it prints the currently remembered file name.

(1,$)g/*RE*/*command list*

In the *g*lobal command, the first step is to mark every line that matches the given RE. Then, for every such line, the given *command list* is executed with **.** initially set to that line. A single command or the first of a list of commands appears on the same line as the global command. All lines of a multi-line list except the last line must be ended with a \; *a*, *i*, and *c* commands and associated input are permitted. The **.** terminating input mode may be omitted if it would be the last line of the *command list*. An empty *command list* is equivalent to the *p* command. The *g*, *G*, *v*, and *V* commands are *not* permitted in the *command list*. See also **BUGS** and the last paragraph before **FILES** below.

(1,$)G/*RE*/

In the interactive *G*lobal command, the first step is to mark every line that matches the given RE. Then, for every such line, that line is printed, **.** is changed to that line, and any *one* command (other than one of the *a*, *c*, *i*, *g*, *G*, *v*, and *V* commands) may be input and is executed. After the execution of that command, the next marked line is printed, and so on; a new-line acts as a null command; an **&** causes the re-execution of the most recent command executed within the current invocation of *G*. Note that the commands input as part of the execution of the *G* command may address and affect *any* lines in the buffer. The *G* command can be terminated by an interrupt signal (ASCII DEL or BREAK).

h

The *h*elp command gives a short error message that explains the reason for the most recent **?** diagnostic.

H

The *H*elp command causes *ed* to enter a mode in which error messages are printed for all subsequent **?** diagnostics. It will also explain the previous **?** if there was one. The *H* command alternately turns this mode on and off; it is initially off.

(.)i
<text>
.

The *i*nsert command inserts the given text before the addressed line; **.** is left at the last inserted line, or, if there were none, at the addressed line. This command differs from the *a* command only in the placement of the input text. Address 0 is not legal for this command. The maximum number of characters that may be entered from a terminal is 256 per line (including the new-line character).

(.,.+1)j

The *j*oin command joins contiguous lines by removing the appropriate new-line characters. If exactly one address is given, this command does nothing.

(.)k*x*

The mar*k* command marks the addressed line with name *x*, which must be an ASCII lower-case letter (a-z). The address '*x* then addresses this line; **.** is unchanged.

(.,.)l

The *l*ist command prints the addressed lines in an unambiguous way: a few non-printing characters (e.g., *tab, backspace*) are represented by visually mnemonic overstrikes. All other non-printing characters are printed in octal, and long lines are folded. An *l* command may be appended to any other command other than *e*,*f*, *r*, or *w*.

(.,.)m*a*

The *m*ove command repositions the addressed line(s) after the line addressed by *a*. Address 0 is legal for *a* and causes the addressed line(s) to be moved to the beginning of the file. It is an error if address *a* falls within the range of moved lines; **.** is left at the last line moved.

(.,.)n

The *n*umber command prints the addressed lines, preceding each line by its line number and a tab character; **.** is left at the last line printed. The *n* command may be appended to any other command other than *e*,*f*, *r*, or *w*.

(.,.)p

The *p*rint command prints the addressed lines; **.** is left at the last line printed. The *p* command may be appended to any other command other than *e*,*f*, *r*, or *w*. For example, *dp* deletes the current line and prints the new current line.

P

The editor will prompt with a * for all subsequent commands. The *P* command alternately turns this mode on and off; it is initially off.

q

The *q*uit command causes *ed* to exit. No automatic write of a file is done; however, see **DIAGNOSTICS**, below.

Q

The editor exits without checking if changes have been made in the buffer since the last *w* command.

($)r *file*

The *r*ead command reads in the given file after the addressed line. If no file name is given, the currently remembered file name, if any, is used (see *e* and *f* commands). The currently remembered file name is *not* changed unless *file* is the very first file name mentioned since *ed* was invoked. Address 0 is legal for *r* and causes the file to be read at the beginning of the buffer. If the read is successful, the number of characters read is typed; **.** is set to the last line read in. If *file* is replaced by !, the rest of the line is taken to be a shell (*sh*(1)) command whose output is to be read. For example, "$r !ls" appends current directory to the end of the file being edited. Such a shell command is *not* remembered as the current file name.

(**.**,**.**)s/*RE*/*replacement*/ or
(**.**,**.**)s/*RE*/*replacement*/**g** or
(**.**,**.**)s/*RE*/*replacement*/**n** n = 1-512

The *s*ubstitute command searches each addressed line for an occurrence of the specified RE. In each line in which a match is found, all (non-overlapped) matched strings are replaced by the *replacement* if the global replacement indicator g appears after the command. If the global indicator does not appear, only the first occurrence of the matched string is replaced. If a number n appears after the command, only the n th occurrence of the matched string on each addressed line is replaced. It is an error for the substitution to fail on *all* addressed lines. Any character other than space or new-line may be used instead of / to delimit the RE and the *replacement*; **.** is left at the last line on which a substitution occurred. See also the last paragraph before FILES below.

An ampersand (&) appearing in the *replacement* is replaced by the string matching the RE on the current line. The special meaning of & in this context may be suppressed by preceding it by \. As a more general feature, the characters *n*, where *n* is a digit, are replaced by the text matched by the *n*-th regular subexpression of the specified RE enclosed between \(and \). When nested parenthesized subexpressions are present, *n* is determined by counting occurrences of \(starting from the left. When the character % is the only character in the *replacement*, the *replacement* used in the most recent substitute command is used as the *replacement* in the current substitute command. The % loses its special meaning when it is in a replacement string of more than one character or is preceded by a \.

A line may be split by substituting a new-line character into it. The new-line in the *replacement* must be escaped by preceding it by \. Such substitution cannot be done as part of a *g* or *v* command list.

(.,.)t*a*

This command acts just like the *m* command, except that a *copy* of the addressed lines is placed after address *a* (which may be 0); . is left at the last line of the copy.

u

The *u*ndo command nullifies the effect of the most recent command that modified anything in the buffer, namely the most recent *a*, *c*, *d*, *g*, *i*, *j*, *m*, *r*, *s*, *t*, *v*, *G*, or *V* command.

(1,$)v/*RE*/*command list*

This command is the same as the global command *g* except that the *command list* is executed with . initially set to every line that does *not* match the RE.

(1,$)V/*RE*/

This command is the same as the interactive global command *G* except that the lines that are marked during the first step are those that do *not* match the RE.

(1,$)w *file*

The *w*rite command writes the addressed lines into the named file. If the file does not exist, it is created with mode 666 (readable and writable by everyone), unless your *umask* setting (see *umask*(1)) dictates otherwise. The currently remembered file name is *not* changed unless *file* is the very first file name mentioned since *ed* was invoked. If no file name is given, the currently remembered file name, if any, is used (see *e* and *f* commands); . is unchanged. If the command is successful, the number of characters written is typed. If *file* is replaced by !, the rest of the line is taken to be a shell (*sh*(1)) command whose standard input is the addressed lines. Such a shell command is *not* remembered as the current file name.

X

A key is prompted for, and it is used in subsequent e, r, and w commands to decrypt and encrypt text using the *crypt*(1) algorithm. An educated guess is made to determine whether text read in for the e and r commands is encrypted. A null key turns off encryption. Subsequent *e*, *r*, and *w* commands will use this key to encrypt or decrypt the text (see *crypt*(1)). An explicitly empty key turns off encryption. Also, see the –x option of *ed*.

($)=
: The line number of the addressed line is typed; **.** is unchanged by this command.

!*shell command*
: The remainder of the line after the **!** is sent to the UNIX system shell (*sh*(1)) to be interpreted as a command. Within the text of that command, the unescaped character **%** is replaced with the remembered file name; if a **!** appears as the first character of the shell command, it is replaced with the text of the previous shell command. Thus, **!!** will repeat the last shell command. If any expansion is performed, the expanded line is echoed; **.** is unchanged.

(.+1)<new-line>
: An address alone on a line causes the addressed line to be printed. A new-line alone is equivalent to **.+1p**; it is useful for stepping forward through the buffer.

If an interrupt signal (ASCII DEL or BREAK) is sent, *ed* prints a **?** and returns to *its* command level.

Some size limitations: 512 characters in a line, 256 characters in a global command list, and 64 characters in the pathname of a file (counting slashes). The limit on the number of lines depends on the amount of user memory: each line takes 1 word.

When reading a file, *ed* discards ASCII NUL characters.

If a file is not terminated by a new-line character, **ed** adds one and puts out a message explaining what it did.

If the closing delimiter of a RE or of a replacement string (e.g., /) would be the last character before a new-line, that delimiter may be omitted, in which case the addressed line is printed. The following pairs of commands are equivalent:

s/s1/s2	s/s1/s2/p
g/s1	g/s1/p
?s1	?s1?

FILES

$TMPDIR	if this environmental variable is not null, its value is used in place of **/usr/tmp** as the directory name for the temporary work file.
/usr/tmp	if **/usr/tmp** exists, it is used as the directory name for the temporary work file.

/tmp	if the environmetal variable **TMPDIR** does not exist or is null, and if **/usr/tmp** does not exist, then **/tmp** is used as the directory name for the temporary work file.
ed.hup	work is saved here if the terminal is hung up.

NOTES

The – option, although it continues to be supported, has been replaced in the documentation by the **–s** option that follows the Command Syntax Standard (see *intro*(1)).

SEE ALSO

edit(1), ex(1), grep(1), sed(1), sh(1), stty(1), umask(1), vi(1).
fspec(4), regexp(5) in the *System Administrator's Reference Manual.*

DIAGNOSTICS

?	for command errors. Type the 'h' command for a longer description of the error.
?*file*	for an inaccessible file. (use the *h*elp and *H*elp commands for detailed explanations).

If changes have been made in the buffer since the last *w* command that wrote the entire buffer, *ed* warns the user if an attempt is made to destroy *ed*'s buffer via the *e* or *q* commands. It prints ? and allows one to continue editing. A second *e* or *q* command at this point will take effect. The –s command-line option inhibits this feature.

WARNINGS

The encryption options and commands are provided with the Security Administration Utilities package, which is available only in the United States.

BUGS

A *!* command cannot be subject to a *g* or a *v* command.

The *!* command and the ! escape from the *e*, *r*, and *w* commands cannot be used if the editor is invoked from a restricted shell (see *sh*(1)).

The sequence \n in a RE does not match a new-line character.

If the editor input is coming from a command file (e.g., ed file < ed-cmd-file), the editor will exit at the first failure.

NAME

edge – window based debugger

SYNOPSIS

edge [**−r**] [**−i**] [**−I** dir] [**−c** file] [**−p** process-id] [**−P** process-name] [**−M** menu-max] [objfile]

DESCRIPTION

Edge is a window-based interface to *dbx*(1). *Objfile* is an object file that you create by using the **-g** flag when you compile your **FORTRAN**, **C**, **Pascal**, or **PL/1** source code. When you start **edge**, it creates three windows: the *dbx*(1) command window, the source window, and the user input/output window. If you do not specify a process with the **−p** or **−P** flag, do not specify an object file, or **edge** cannot find the source for the program specified, the source window will not appear. The **dbx** window contains the *dbx*(1) prompt. You can use the typical *dbx*(1) commands here. This window also contains buttons for sending commands to *dbx*(1). The top pane of this window lists all of the processes running on the system owned by you. All of these processes can be debugged. The user window, stdin/stdout/stderr, acts as the user program's input and output window. The source window contains the user program's source code, and the buttons that take arguments selected with the mouse button in the source window. You can resize, reshape and move these windows by pressing and holding the right mouse button in the title bar. This produces the *4sight*(1) menu. See below for more information on windows.

Arguments to **edge** are the same as those to *dbx*(1). The **−M** option permits the user to specify the maximum number of files, *menu-max*, that *edge* will display on the "File" and "Edit" menus. The default and absolute limit is 500 files. Because, *dbx* checks for the existence of each of these files before passing them to *edge* it is possible that this checking will extraordinarily slow down the start up of *edge*. The user may specify any number of files for the menus to handle between 0 and the absolute limit, 500.

The Dbx Window

See *dbx*(1) for the line interface in this window. The top pane lists all of the processes running on the system owned by you. The **addproc**, **delproc**, **activate**, and **debug** buttons take arguments selected in the status pane of the dbx window. Select a process by holding the left mouse button down over the line of the process to be added, selected, etc. You use the buttons in this window by moving the cursor over the item you want to select, then pressing the left mouse button. The buttons invoke these commands:

addproc

Add the selected process to the process pool. The process will not be stopped until you explicitly suspend it.

delproc Delete the selected process from the process pool.

suspend
Stop the active process.

activate
Make the selected process the active process.

debug Add the selected process to the process pool, suspend it and make it the active process.

The rest of the functions in this window do not take arguments. They are:

rerun Sends the **rerun** command to **dbx**(1). **Edge** uses arguments you specified for the last **run** command to rerun your program.

cont Continues execution.

step Single steps one source line.

next Single steps one source line, but skips over function and subroutine calls.

where Prints a stack trace from the current point of execution.

trace Traces execution of all source lines. The currently executing line is highlighted in green in the source window.

interrupt
Sends an interrupt to *dbx*(1).

sh Forks a shell in a *wsh*(1) window.

quit Exits from **edge**.

When you press the the right mouse button over any **edge** window, you see a menu with the following choices:

attach Attaches input focus to the window.

select Pops and attaches to the window.

edit Lets you edit a source file. This choice contains a sub-menu that lists the source files that make up your program. Slide the cursor to the right to see the sub-menu, then select the file you want to edit. This invokes the editor specified by the $EDIT environment variable on the selected file.

file Lets you read another source file into the source window. This choice contains a sub-menu that lists the source files that make up your program. Select the file you want displayed in the source window.

The Source Window

The source window displays your source. The currently executing line is highlighted in green. Breakpoints are highlighted in red. Selected text is highlighted in yellow. Select text by pressing and holding the left mouse button, dragging the cursor over the text to be selected, then releasing the mouse button. When you use the **file** or / commands in the **dbx**(1) window, the results will be reflected in the source window. The title bar of this window contains the name of the file being displayed. The buttons in this window use selected text or line numbers as arguments. The buttons are:

print	Prints the value of the variable selected in the source window.
print *	Prints the value pointed to by the variable selected in the source window.
px	Prints the value of the variable selected in the source window in hexadecimal.
stop at	Sets a breakpoint at the currently selected line in the source window.
stop in	Sets a breakpoint in the currently selected function in the source window.
cont to	Continue execution to the line selected in the source window.
edit	Edit the currently selected function.
list	List the currently selected function in the source window.

The User window.

This window acts as the user program's stdin/stdout/stderr.

Edgerc syntax.

If the file **.edgerc** exists in the current directory, and/or in the home directory, **edge** reads it. The syntax for commands in the **.edgerc** file follow:

size window [columns] [rows]

window specifies either the dbx, source, or user window. *Rows* specifies the number of rows the window should occupy, and *columns* specifies the number of columns the window should occupy. If you do not specify *rows* and *columns*, you will be prompted to size and place your windows.

origin window [xcoord] [ycoord]

This command causes the specified window's origin to be at screen coordinates *xcoord, ycoord.* If you do not specify *xcoord* and *ycoord*, you will be prompted to place your own windows.

color text background
: *Text* and *background* are integers that specify the color indices to be used for text and background.

FILES

a.out	object file
.edgerc	initialization commands
$HOME/.wshttymode	custom terminal settings

SEE ALSO

cc(1), f77(1), pc(1), pl1(1), dbx(1), wsh(1G)
''Getting Started with edge for C''
''Getting Started with edge for FORTRAN''
''Getting Started with edge for Pascal''

BUGS

edge uses the color indices 0-7. If you modify these, you modify **edge's** colors.

edge is slow to start up if there are many files comprising the object to be debugged. The –M option described above can be used to specify the maximum number of files *edge* will attempt to verify before giving up.

NAME

edit – text editor (variant of ex for casual users)

SYNOPSIS

edit [–**r**] [–**x**] [–**C**] *name*...

DESCRIPTION

edit is a variant of the text editor *ex* recommended for new or casual users who wish to use a command-oriented editor. It operates precisely as *ex*(1) with the following options automatically set:

novice	ON
report	ON
showmode	ON
magic	OFF

These options can be turned on or off via the **set** command in *ex*(1).

–**r** Recover file after an editor or system crash.

–**x** Encryption option; when used the file will be encrypted as it is being written and will require an encryption key to be read. *edit* makes an educated guess to determine if a file is encrypted or not. See *crypt*(1). Also, see the **WARNING** section at the end of this manual page.

–**C** Encryption option; the same as –**x** except that *edit* assumes files are encrypted.

The following brief introduction should help you get started with *edit*. If you are using a CRT terminal you may want to learn about the display editor *vi*.

To edit the contents of an existing file you begin with the command **edit** *name* to the shell. *edit* makes a copy of the file that you can then edit, and tells you how many lines and characters are in the file. To create a new file, you also begin with the command **edit** with a filename: **edit** *name*; the editor will tell you it is a [New File].

The *edit* command prompt is the colon (**:**), which you should see after starting the editor. If you are editing an existing file, then you will have some lines in *edit's* buffer (its name for the copy of the file you are editing). When you start editing, *edit* makes the last line of the file the current line. Most commands to *edit* use the current line if you do not tell them which line to use. Thus if you say **print** (which can be abbreviated **p**) and type carriage return (as you should after all *edit* commands), the current line will be printed. If you **delete** (**d**) the current line, *edit* will print the new current line, which is usually the next line in the file. If you **delete** the last line, then the new last line becomes the current one.

If you start with an empty file or wish to add some new lines, then the **append** (**a**) command can be used. After you execute this command (typing a carriage return after the word **append**), *edit* will read lines from your terminal until you type a line consisting of just a dot (.); it places these lines after the current line. The last line you type then becomes the current line. The command **insert** (**i**) is like **append**, but places the lines you type before, rather than after, the current line.

edit numbers the lines in the buffer, with the first line having number 1. If you execute the command **1**, then *edit* will type the first line of the buffer. If you then execute the command **d**, *edit* will delete the first line, line 2 will become line 1, and *edit* will print the current line (the new line 1) so you can see where you are. In general, the current line will always be the last line affected by a command.

You can make a change to some text within the current line by using the **substitute** (**s**) command: s/*old*/*new*/ where *old* is the string of characters you want to replace and *new* is the string of characters you want to replace *old* with.

The command **file** (**f**) will tell you how many lines there are in the buffer you are editing and will say [Modified] if you have changed the buffer. After modifying a file, you can save the contents of the file by executing a **write** (**w**) command. You can leave the editor by issuing a **quit** (**q**) command. If you run *edit* on a file, but do not change it, it is not necessary (but does no harm) to **write** the file back. If you try to **quit** from *edit* after modifying the buffer without writing it out, you will receive the message No write since last change (:quit! overrides), and *edit* will wait for another command. If you do not want to write the buffer out, issue the **quit** command followed by an exclamation point (**q!**). The buffer is then irretrievably discarded and you return to the shell.

By using the **d** and **a** commands and giving line numbers to see lines in the file, you can make any changes you want. You should learn at least a few more things, however, if you will use *edit* more than a few times.

The **change** (**c**) command changes the current line to a sequence of lines you supply (as in **append**, you type lines up to a line consisting of only a dot (.). You can tell **change** to change more than one line by giving the line numbers of the lines you want to change, i.e., **3,5c**. You can print lines this way too: **1,23p** prints the first 23 lines of the file.

The **undo** (**u**) command reverses the effect of the last command you executed that changed the buffer. Thus if you execute a **substitute** command that does not do what you want, type **u** and the old contents of the line will be restored. You can also **undo** an **undo** command. *edit* will give you a warning message when a command affects more than one line of the buffer. Note that commands such as **write** and **quit** cannot be undone.

To look at the next line in the buffer, type carriage return. To look at a number of lines, type **^D** (while holding down the control key, press **d**) rather than carriage return. This will show you a half-screen of lines on a CRT or 12 lines on a hardcopy terminal. You can look at nearby text by executing the **z** command. The current line will appear in the middle of the text displayed, and the last line displayed will become the current line; you can get back to the line where you were before you executed the **z** command by typing **´´**. The **z** command has other options: **z–** prints a screen of text (or 24 lines) ending where you are; **z+** prints the next screenful. If you want less than a screenful of lines, type **z.11** to display five lines before and five lines after the current line. (Typing **z.***n*, when *n* is an odd number, displays a total of *n* lines, centered about the current line; when *n* is an even number, it displays *n*–1 lines, so that the lines displayed are centered around the current line.) You can give counts after other commands; for example, you can delete 5 lines starting with the current line with the command **d5** .

To find things in the file, you can use line numbers if you happen to know them; since the line numbers change when you insert and delete lines this is somewhat unreliable. You can search backwards and forwards in the file for strings by giving commands of the form */text/* to search forward for *text* or *?text?* to search backward for *text* . If a search reaches the end of the file without finding *text*, it wraps around and continues to search back to the line where you are. A useful feature here is a search of the form */^text/* which searches for *text* at the beginning of a line. Similarly */text$/* searches for *text* at the end of a line. You can leave off the trailing / or ? in these commands.

The current line has the symbolic name dot (.); this is most useful in a range of lines as in **.,$p** which prints the current line plus the rest of the lines in the file. To move to the last line in the file, you can refer to it by its symbolic name $. Thus the command **$d** deletes the last line in the file, no matter what the current line is. Arithmetic with line references is also possible. Thus the line **$–5** is the fifth before the last and **.+20** is 20 lines after the current line.

You can find out the current line by typing **.=**. This is useful if you wish to move or copy a section of text within a file or between files. Find the first and last line numbers you wish to copy or move. To move lines 10 through 20, type **10,20d a** to delete these lines from the file and place them in a buffer named **a**. *edit* has 26 such buffers named **a** through **z**. To put the contents of buffer **a** after the current line, type **put a**. If you want to move or copy these lines to another file, execute an **edit** (**e**) command after copying the lines; following the **e** command with the name of the other file you wish to edit, i.e., **edit chapter2**. To copy lines without deleting them, use **yank** (**y**) in place of **d**. If the text you wish to move or copy is all within one file, it is not necessary to use named buffers. For example, to move lines 10 through 20 to the end of the file, type **10,20m $**.

SEE ALSO

ed(1), ex(1), vi(1).

WARNING

The encryption options are provided with the Security Administration Utilities package, which is available only in the United States.

NAME

egrep – search a file for a pattern using full regular expressions

SYNOPSIS

egrep [options] full regular expression [file ...]

DESCRIPTION

egrep (*expression grep*) searches files for a pattern of characters and prints all lines that contain that pattern. *egrep* uses full regular expressions (expressions that have string values that use the full set of alphanumeric and special characters) to match the patterns. It uses a fast deterministic algorithm that sometimes needs exponential space.

egrep accepts full regular expressions as in *ed*(1), except for \(and \), with the addition of:

1. A full regular expression followed by + that matches one or more occurrences of the full regular expression.
2. A full regular expression followed by ? that matches 0 or 1 occurrences of the full regular expression.
3. Full regular expressions separated by | or by a new-line that match strings that are matched by any of the expressions.
4. A full regular expression that may be enclosed in parentheses () for grouping.

Be careful using the characters $, *, [, ^, | , (,), and \ in *full regular expression*, because they are also meaningful to the shell. It is safest to enclose the entire *full regular expression* in single quotes '...'.

The order of precedence of operators is [ ], then * ? +, then concatenation, then | and new-line.

If no files are specified, *egrep* assumes standard input. Normally, each line found is copied to the standard output. The file name is printed before each line found if there is more than one input file.

Command line options are:

–b	Precede each line by the block number on which it was found. This can be useful in locating block numbers by context (blocks are 512 bytes long and number from 0).
–c	Print only a count of the lines that contain the pattern.
–i	Ignore upper/lower case distinction during comparisons.
–l	Print the names of files with matching lines once, separated by new-lines. Does not repeat the names of files when the pattern is found more than once.

−n Precede each line by its line number in the file (first line is 1).

−s Silent mode. No pattern matches or error messages are printed. This option allows command expressions to check *egrep*'s exit status without having to deal with output.

−v Print all lines except those that contain the pattern.

−e *special_expression*
Search for a *special expression* (*full regular expression* that begins with a −).

−f *file* Take the list of *full regular expressions* from *file*.

SEE ALSO

ed(1), fgrep(1), grep(1), sed(1), sh(1).

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files (even if matches were found).

BUGS

Ideally there should be only one *grep* command, but there is not a single algorithm that spans a wide enough range of space-time tradeoffs. Lines are limited to BUFSIZ characters; longer lines are truncated. BUFSIZ is defined in **/usr/include/stdio.h.**

NAME

enable, disable – enable/disable LP printers

SYNOPSIS

enable printers
disable [**–c**] [**–r**[reason]] printers

DESCRIPTION

enable activates the named *printers*, enabling them to print requests taken by *lp*(1). Use *lpstat*(1) to find the status of printers.

Disable deactivates the named *printers*, disabling them from printing requests taken by *lp*(1). By default, any requests that are currently printing on the designated printers will be reprinted in their entirety either on the same printer or on another member of the same class. Use *lpstat*(1) to find the status of printers. Options useful with *disable* are:

–c	Cancel any requests that are currently printing on any of the designated printers.
–r[*reason*]	Associates a *reason* with the deactivation of the printers. This reason applies to all printers mentioned up to the next **–r** option. If the **–r** option is not present or the **–r** option is given without a reason, then a default reason will be used. *Reason* is reported by *lpstat*(1).

FILES

/usr/spool/lp/*

SEE ALSO

lp(1), lpstat(1).

NAME

env – set environment for command execution

SYNOPSIS

env [–] [name=value] ... [command args]

DESCRIPTION

env obtains the current *environment*, modifies it according to its arguments, then executes the command with the modified environment. Arguments of the form *name=value* are merged into the inherited environment before the command is executed. The – flag causes the inherited environment to be ignored completely, so that the command is executed with exactly the environment specified by the arguments.

If no command is specified, the resulting environment is printed, one name-value pair per line.

SEE ALSO

sh(1).
exec(2), profile(4), environ(5) in the *Programmer's Reference Manual*.

NAME

ex – text editor

SYNOPSIS

ex [**–s**] [**–v**] [**–t** tag] [**–r** file] [**–L**] [**–R**] [**–x**] [**–C**] [**–c** command] file ...

DESCRIPTION

ex is the root of a family of editors: *ex* and *vi*. *ex* is a superset of *ed*, with the most notable extension being a display editing facility. Display based editing is the focus of *vi*.

If you use a window or a CRT terminal, you may wish to use the *vi*(1) editor, which focuses on the display-editing portion of *ex*.

For ed Users

If you have used *ed*(1) you will find that, in addition to having all of the *ed*(1) commands available, *ex* has a number of additional features useful on CRT terminals. Intelligent terminals and high speed terminals are very pleasant to use with *vi*. Generally, the *ex* editor uses far more of the capabilities of terminals than *ed*(1) does, and uses the terminal capability data base (see *terminfo*(4)) and the type of the terminal you are using from the environmental variable TERM to determine how to drive your terminal efficiently. The editor makes use of features such as insert and delete character and line in its **visual** command (which can be abbreviated **vi**) and which is the central mode of editing when using *vi*(1).

ex contains a number of features for easily viewing the text of the file. The **z** command gives easy access to windows of text. Typing **^D** (control-d) causes the editor to scroll a half-window of text and is more useful for quickly stepping through a file than just typing return. Of course, the screen-oriented **visual** mode gives constant access to editing context.

ex gives you help when you make mistakes. The **undo** (**u**) command allows you to reverse any single change which goes astray. *ex* gives you a lot of feedback, normally printing changed lines, and indicates when more than a few lines are affected by a command so that it is easy to detect when a command has affected more lines than it should have.

The editor also normally prevents overwriting existing files, unless you edited them, so that you do not accidentally overwrite a file other than the one you are editing. If the system (or editor) crashes, or you accidentally hang up the telephone, you can use the editor **recover** command (or **–r** *file* option) to retrieve your work. This will get you back to within a few lines of where you left off.

ex has several features for dealing with more than one file at a time. You can give it a list of files on the command line and use the **next** (**n**) command to deal with each in turn. The **next** command can also be given a list of file names, or a pattern as used by the shell to specify a new set of files to be dealt with. In general, file names in the editor may be formed with full shell metasyntax. The metacharacter '%' is also available in forming file names and is replaced by the name of the current file. For editing large groups of related files you can use *ex's* **tag** command to quickly locate functions and other important points in any of the files. This is useful when working on a large program when you want to quickly find the definition of a particular function. The *ctags*(1) utility builds a *tags* file for C, Pascal, and FORTRAN programs.

The editor has a group of buffers whose names are the ASCII lower-case letters (**a-z**). You can place text in these named buffers where it is available to be inserted elsewhere in the file. The contents of these buffers remain available when you begin editing a new file using the **edit** (**e**) command.

There is a command & in *ex* which repeats the last **substitute** command. In addition, there is a confirmed substitute command. You give a range of substitutions to be done and the editor interactively asks whether each substitution is desired.

It is possible to ignore the case of letters in searches and substitutions. *ex* also allows regular expressions which match words to be constructed. This is convenient, for example, in searching for the word "edit" if your document also contains the word "editor."

ex has a set of options which you can set to tailor it to your liking. One option which is very useful is the **autoindent** option that allows the editor to supply leading white space to align text automatically. You can then use ^D as a backtab and space or tab to move forward to align new code easily.

Miscellaneous useful features include an intelligent **join** (**j**) command that supplies white space between joined lines automatically, commands "<" and ">" which shift groups of lines, and the ability to filter portions of the buffer through commands such as *sort*(1).

Invocation Options

The following invocation options are interpreted by *ex* (previously documented options are discussed in the **NOTES** section at the end of this manual page):

−s	Suppress all interactive-user feedback. This is useful in processing editor scripts.

−v	Invoke *vi*
−t *tag*	Edit the file containing the *tag* and position the editor at its definition.
−r *file*	Edit *file* after an editor or system crash. (Recovers the version of *file* that was in the buffer when the crash occurred.)
−L	List the names of all files saved as the result of an editor or system crash.
−R	**Readonly** mode; the **readonly** flag is set, preventing accidental overwriting of the file.
−x	Encryption option; when used, *ex* simulates an **X** command and prompts the user for a key. This key is used to encrypt and decrypt text using the algorithm of *crypt*(1). The **X** command makes an educated guess to determine whether text read in is encrypted or not. The temporary buffer file is encrypted also, using a transformed version of the key typed in for the **−x** option. See *crypt*(1). Also, see the **WARNINGS** section at the end of this manual page.
−C	Encryption option; the same as the **−x** option, except that *ex* simulates a **C** command. The **C** command is like the **X** command, except that all text read in is assumed to have been encrypted.
−c *command*	Begin editing by executing the specified editor *command* (usually a search or positioning command).

The *file* argument indicates one or more files to be edited.

ex States

Command	Normal and initial state. Input prompted for by **:**. Your line kill character cancels a partial command.
Insert	Entered by **a**, **i**, or **c**. Arbitrary text may be entered. Insert state normally is terminated by a line having only "**.**" on it, or, abnormally, with an interrupt.
Visual	Entered by typing **vi**; terminated by typing **Q** or **^** (control-\\).

ex Command Names and Abbreviations

abbrev	**ab**	**map**		**set**	**se**
append	**a**	**mark**	**ma**	**shell**	**sh**

args	**ar**	**move**	**m**	**source**	**so**
change	**c**	**next**	**n**	**substitute**	**s**
copy	**co**	**number**	**nu**	**unabbrev**	**unab**
delete	**d**	**preserve**	**pre**	**undo**	**u**
edit	**e**	**print**	**p**	**unmap**	**unm**
file	**f**	**put**	**pu**	**version**	**ve**
global	**g**	**quit**	**q**	**visual**	**vi**
insert	**i**	**read**	**r**	**write**	**w**
join	**j**	**recover**	**rec**	**xit**	**x**
list	**l**	**rewind**	**rew**	**yank**	**ya**

ex Commands

shell escape	!
forced encryption	C
heuristic encryption	X
lshift	<
print next	CR
resubst	&
rshift	>
scroll	^D
window	z

ex Command Addresses

n	line *n*	/*pat*	next with *pat*
.	current	?*pat*	previous with *pat*
$	last	*x-n*	*n* before *x*
+	next	*x,y*	*x* through *y*
–	previous	´*x*	marked with *x*
+*n*	*n* forward	´´	previous context
%	1,$		

Initializing options

EXINIT	place set's here in environment variable
$HOME/.exrc	editor initialization file
./.exrc	editor initialization file
set *x*	enable option *x*
set no*x*	disable option *x*
set *x*=*val*	give value *val* to option *x*
set	show changed options
set all	show all options
set *x*?	show value of option *x*

If the **EXINIT** environment variable is set, initialization options are taken from that variable. Otherwise, initialization options are taken from **$HOME/.exrc**, if it exists. Finally, if the **exrc** option is set (either by **EXINIT** or **$HOME/.exrc**,) then initialization options will be taken from

./.exrc, if it exists.

Most useful options and their abbreviations

autoindent	**ai**	supply indent
autowrite	**aw**	write before changing files
directory		pathname of directory for temporary work files
exrc	**ex**	allow *vi/ex* to read the **.exrc** in the current directory. This option is set in the **EXINIT** shell variable or in the **.exrc** file in the **$HOME** directory.
ignorecase	**ic**	ignore case of letters in scanning
list		print ^I for tab, $ at end
magic		treat **.** [* special in patterns
modelines		first five lines and last five lines executed as *vi/ex* commands if they are of the form **ex:***command*: or **vi:***command*:
number	**nu**	number lines
paragraphs	**para**	macro names that start paragraphs
redraw		simulate smart terminal
report		informs you if the number of lines modified by the last command is greater than the value of the **report** variable
scroll		command mode lines
sections	**sect**	macro names that start sections
shiftwidth	**sw**	for < >, and input ^D
showmatch	**sm**	to) and } as typed
showmode	**smd**	show insert mode in *vi*
slowopen	**slow**	stop updates during insert
term		specifies to **vi** the type of terminal being used (the default is the value of the environmental variable TERM)
window		visual mode lines
wrapmargin	**wm**	automatic line splitting
wrapscan	**ws**	search around end (or beginning) of buffer

Scanning pattern formation

^	beginning of line
$	end of line
.	any character
\<	beginning of word
\>	end of word
[*str*]	any character in *str*
[^*str*]	any character not in *str*

[*x*–*y*]	any character between *x* and *y*
*	any number of preceding characters

FILES

/usr/lib/exrecover	recover command
/usr/lib/expreserve	preserve command
/usr/lib/terminfo/*	describes capabilities of terminals
$HOME/.exrc	editor startup file
./.exrc	editor startup file
/tmp/Ex*nnnnn*	editor temporary
/tmp/Rx*nnnnn*	named buffer temporary
/usr/preserve/*login*	preservation directory (where *login* is the user's login name)

NOTES

Several options, although they continue to be supported, have been replaced in the documentation by options that follow the Command Syntax Standard (see *intro*(1)). The – option has been replaced by –**s**, a –**r** option that is not followed with an option-argument has been replaced by –L, and +*command* has been replaced by –**c** *command*.

SEE ALSO

crypt(1), ctags(1), ed(1), edit(1), grep(1), sed(1), sort(1), vi(1).
curses(3X), in the *Programmer's Reference Manual.*
term(4), terminfo(4) in the *System Administrator's Reference Manual.*
User's Guide.
curses/terminfo chapter of the *Programmer's Guide*.

WARNINGS

The encryption options and commands are provided with the Security Administration Utilities package, which is available only in the United States.

BUGS

The z command prints the number of logical rather than physical lines. More than a screen full of output may result if long lines are present.

File input/output errors do not print a name if the command line –s option is used.

There is no easy way to do a single scan ignoring case.

The editor does not warn if text is placed in named buffers and not used before exiting the editor.

Null characters are discarded in input files and cannot appear in resultant files.

NAME

exporttonews – Pass a login shell's environment to the NeWS server

SYNOPSIS

exporttonews

DESCRIPTION

exporttonews is run automatically by the NeWS server at startup. It forks a login shell indicated by the SHELL environment variable, and then dumps the environment variables from that shell into the NeWS server.

The exact commands passed to the shell vary depending on what shell is used. If the shell is the C-Shell, the command line is:

```
source .login ; echo ::: ENV ; exec /bin/env
```

Otherwise, a command line which works with bourne and korn shells is sent:

```
echo ::: ENV ; exec /bin/env
```

If you are using a shell which is not compatible with /bin/sh, you can have *exporttonews* executive a custom command line to dump the environment. The file /etc/env/$SHELL is searched for and if it exists, that command is executed to dump the environment, instead of the default. It is important to include the "`echo ::: env`" just before the shell dumps the environment, as this is what *exporttonews* looks for to signal the beginning of the environment.

BUGS

If the environment established at login time has a variable containing a newline, *exporttonews* will probably fail to pass the full value of that variable to the NeWS server.

The login shell forked by *exporttonews* does not have a tty open, so some commands (such as tty(1)) may return strange results. To alleviate this problem, *exporttonews* sets an environment variable "ENVONLY", and login scripts which depend on having programs like tty(1) work, can skip those commands if the variable ENVONLY is set.

NAME

expr – evaluate arguments as an expression

SYNOPSIS

expr arguments

DESCRIPTION

The arguments are taken as an expression. After evaluation, the result is written on the standard output. Terms of the expression must be separated by blanks. Characters special to the shell must be escaped. Note that **0** is returned to indicate a zero value, rather than the null string. Strings containing blanks or other special characters should be quoted. Integer-valued arguments may be preceded by a unary minus sign. Internally, integers are treated as 32-bit, 2s complement numbers.

The operators and keywords are listed below. Characters that need to be escaped are preceded by \. The list is in order of increasing precedence, with equal precedence operators grouped within { } symbols.

expr \| *expr*
: returns the first *expr* if it is neither null nor **0**, otherwise returns the second *expr*.

expr \& *expr*
: returns the first *expr* if neither *expr* is null or **0**, otherwise returns **0**.

expr { =, \>, \>=, \<, \<=, != } *expr*
: returns the result of an integer comparison if both arguments are integers, otherwise returns the result of a lexical comparison.

expr { +, – } *expr*
: addition or subtraction of integer-valued arguments.

expr { *, /, % } *expr*
: multiplication, division, or remainder of the integer-valued arguments.

expr **:** *expr*
: The matching operator **:** compares the first argument with the second argument which must be a regular expression. Regular expression syntax is the same as that of *ed*(1), except that all patterns are "anchored" (i.e., begin with ^) and, therefore, ^ is not a special character, in that context. Normally, the matching operator returns the number of characters matched (**0** on failure). Alternatively, the \(...\) pattern symbols can be used to return a portion of the first argument.

match *expr expr*

The **match** operator is equivalent to the matching operator **:**, except that it is a prefix operator, rather than an infix operator.

EXAMPLES

1.
```
a=`expr $a + 1`
```
adds 1 to the shell variable **a**.

2.
```
# ´For $a equal to either "/usr/abc/file" or just "file"´
expr $a : ´.*/\(.*\)´ \| $a
```
returns the last segment of a path name (i.e., file). Watch out for / alone as an argument: *expr* will take it as the division operator (see BUGS below).

3.
```
# A better representation of example 2.
expr //$a : ´.*/\(.*\)´
```
The addition of the // characters eliminates any ambiguity about the division operator and simplifies the whole expression.

4.
```
expr $VAR : ´.*´
```
returns the number of characters in **$VAR**.

SEE ALSO

ed(1), sh(1).

DIAGNOSTICS

As a side effect of expression evaluation, *expr* returns the following exit values:

0	if the expression is neither null nor **0**
1	if the expression *is* null or **0**
2	for invalid expressions.

syntax error	for operator/operand errors
non-numeric argument	if arithmetic is attempted on such a string

BUGS

After argument processing by the shell, *expr* cannot tell the difference between an operator and an operand except by the value. If $a is an =, the command:

```
expr $a = ´=´
```

looks like:

```
expr = = =
```

as the arguments are passed to *expr* (and they will all be taken as the = operator). The following works:

```
expr X$a = X=
```

NAME

extcentry – extract FORTRAN-callable entry points from a C file

SYNOPSIS

extcentry infile outfile

DESCRIPTION

Extcentry is used to extract C functions for which FORTRAN-callable interface routines (*wrappers*) are to be generated by the program *mkf2c*(1). *Extcentry* accepts as input any C file (*infile*), and outputs (to *outfile*) only those portions which are surrounded by the special comments /* *CENTRY* */ and /* *ENDCENTRY* */.

The first step in generating a FORTRAN-to-C interface routine is to surround only those C functions for which entry points are to be generated by these special comments and to run the file through *extcentry*. The FORTRAN-to-C interface generator program *mkf2c* can then be invoked on the resultant file to generate the assembly language *wrapper*. This is necessary since *mkf2c* understands only a limited subset of the C grammar, and cannot parse such constructs as external declarations, typedefs, and C-preprocessor directives.

SEE ALSO

mkf2c(1)

NAME

factor – obtain the prime factors of a number

SYNOPSIS

factor [integer]

DESCRIPTION

When you use *factor* without an argument, it waits for you to give it an integer. After you give it a positive integer less than or equal to 10^{14}, it factors the integer, prints its prime factors the proper number of times, and then waits for another integer. *factor* exits if it encounters a zero or any non-numeric character.

If you invoke *factor* with an argument, it factors the integer as described above, and then it exits.

The maximum time to factor an integer is proportional to $\sqrt{n}$. *factor* will take this time when n is prime or the square of a prime.

DIAGNOSTICS

factor prints the error message, "Ouch," for input out of range. *factor* prints the error message ‘‘Q: undefined variable’’ for garbage input.

NAME

fgrep – search a file for a character string

SYNOPSIS

fgrep [options] string [file ...]

DESCRIPTION

fgrep (fast *grep*) searches files for a character string and prints all lines that contain that string. *fgrep* is different from *grep(1)* and *egrep(1)* because it searches for a string, instead of searching for a pattern that matches an expression. It uses a fast and compact algorithm.

The characters $, *, [, ^, | , (,), and \ are interpreted literally by *fgrep*, that is, *fgrep* does not recognize full regular expressions as does *egrep*. Since these characters have special meaning to the shell, it is safest to enclose the entire *string* in single quotes ′...′.

If no files are specified, *fgrep* assumes standard input. Normally, each line found is copied to the standard output. The file name is printed before each line found if there is more than one input file.

Command line options are:

–b	Precede each line by the block number on which it was found. This can be useful in locating block numbers by context (blocks are 512 bytes long and number from 0).
–c	Print only a count of the lines that contain the pattern.
–i	Ignore upper/lower case distinction during comparisons.
–l	Print the names of files with matching lines once, separated by new-lines. Does not repeat the names of files when the pattern is found more than once.
–n	Precede each line by its line number in the file (first line is 1).
–s	Silent mode. No pattern matches or error messages are printed. This option allows command expressions to check *fgrep*'s exit status without having to deal with output.
–v	Print all lines except those that contain the pattern.
–x	Print only lines matched entirely.
–e *special_string*	Search for a *special string* (*string* begins with a –).
–f *file*	Take the list of *strings* from *file*.

SEE ALSO

ed(1), egrep(1), grep(1), sed(1), sh(1).

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files (even if matches were found).

BUGS

Ideally there should be only one *grep* command, but there is not a single algorithm that spans a wide enough range of space-time tradeoffs. Lines are limited to BUFSIZ characters; longer lines are truncated. BUFSIZ is defined in **/usr/include/stdio.h.**

NAME

file – determine file type

SYNOPSIS

file [**–c**] [**–f** ffile] [**–m** mfile] arg ...

DESCRIPTION

file performs a series of tests on each argument in an attempt to classify it. If an argument appears to be ASCII, *file* examines the first 512 bytes and tries to guess its language. If an argument is an executable **a.out**, *file* will print the version stamp, provided it is greater than 0.

–c The –c option causes *file* to check the magic file for format errors. This validation is not normally carried out for reasons of efficiency. No file typing is done under **–c**.

–f If the **–f** option is given, the next argument is taken to be a file containing the names of the files to be examined.

–m The **–m** option instructs *file* to use an alternate magic file.

file uses the file **/etc/magic** to identify files that have some sort of *magic number*, that is, any file containing a numeric or string constant that indicates its type. Commentary at the beginning of **/etc/magic** explains its format.

FILES

/etc/magic

SEE ALSO

filehdr(4) in the *Programmer's Reference Manual.*

NAME

find – find files

SYNOPSIS

find path-name-list expression

DESCRIPTION

find recursively descends the directory hierarchy for each path name in the *path-name-list* (that is, one or more path names) seeking files that match a boolean *expression* written in the primaries given below. In the descriptions, the argument *n* is used as a decimal integer where +*n* means more than *n*, −*n* means less than *n* and *n* means exactly *n*. Valid expressions are:

−name *file*	True if *file* matches the current file name. Normal shell argument syntax may be used if escaped (watch out for [, ? and *).
−perm [−]*onum*	True if the file permission flags exactly match the octal number *onum* (see *chmod*(1)). If *onum* is prefixed by a minus sign, only the bits that are set in *onum* are compared with the file permission flags, and the expression evaluates true if they match.
−type *c*	True if the type of the file is *c*, where *c* is **b**, **c**, **d**, **l**, **p**, **f**, or s for block special file, character special file, directory, symbolic link, fifo (a.k.a named pipe), plain file, or socket respectively.
−links *n*	True if the file has *n* links.
−user *uname*	True if the file belongs to the user *uname*. If *uname* is numeric and does not appear as a login name in the /etc/passwd file, it is taken as a user ID.
−group *gname*	True if the file belongs to the group *gname*. If *gname* is numeric and does not appear in the **/etc/group** file, it is taken as a group ID.
−size *n*[**c**]	True if the file is *n* blocks long (512 bytes per block). If *n* is followed by a **c**, the size is in characters.
−inum *n*	True if *n* is the inode number of the file.
−atime *n*	True if the file has been accessed in *n* days (see *stat*(2) for a description of which file operations change the access time of a file). The access time of directories in *path-name-list* is changed by *find* itself.

−mtime *n*	True if the file has been modified in *n* days (see *stat*(2) for a description of which file operations change the modification time of a file).
−ctime *n*	True if the file has been changed in *n* days (see *stat*(2) for a description of which file operations change the change time of a file).
−exec *cmd*	True if the executed *cmd* returns a zero value as exit status. The end of *cmd* must be punctuated by an escaped semicolon. A command argument {} is replaced by the current path name.
−ok *cmd*	Like **−exec** except that the generated command line is printed with a question mark first, and is executed only if the user responds by typing **y**.
−print	Always true; causes the current path name to be printed.
−cpio *device*	Always true; write the current file on *device* in *cpio* (1) format (5120-byte records).
−newer *file*	True if the current file has been modified more recently than the argument *file* (see *stat*(2) for a description of which file operations change the modification time of a file).
−anewer *file*	True if current file has been accessed more recently than the argument *file* (see *stat*(2) for a description of which file operations change the access time of a file).
−cnewer *file*	True if current file has been changed more recently than the argument *file* (see *stat*(2) for a description of which file operations change the change time of a file).
−depth	Always true; causes descent of the directory hierarchy to be done so that all entries in a directory are acted on before the directory itself. This can be useful when *find* is used with *cpio*(1) to transfer files that are contained in directories without write permission.
−prune	Always true; has the side effect of pruning the search tree at the current file. If the current path name is a directory, *find* will not descend into that directory.
−mount	Always true; restricts the search to the file system containing the directory specified, or if no directory was specified, the current directory.

–local	True if the file physically resides on the local system.
–follow	Always true; causes the underlying file of a symbolic link to be checked rather than the symbolic link itself.
(*expression*)	True if the parenthesized expression is true (parentheses are special to the shell and must be escaped).

The primaries may be combined using the following operators (in order of decreasing precedence):

1) The negation of a primary (**!** is the unary *not* operator).

2) Concatenation of primaries (the *and* operation is implied by the juxtaposition of two primaries).

3) Alternation of primaries (**–o** is the *or* operator).

EXAMPLES

To remove all files named **a.out** or ***.o** that have not been accessed for a week:

```
find / \( –name a.out –o –name '*.o' \) –atime +7 –exec rm {} \;
```

To display all character special devices on the root file system except those under any **dev** directory:

```
find / –mount \( –type d –name dev –prune \) –o –type c –print
```

FILES

/etc/passwd	UID information supplier
/etc/group	GID information supplier

SEE ALSO

chmod(1), cpio(1), sh(1), test(1).
stat(2), umask(2), fs(4) in the *Programmer's Reference Manual.*

BUGS

find / –depth always fails with the message: "find: stat failed: : No such file or directory".

NAME

finger – user information lookup program

SYNOPSIS

finger [**–bfhilmpqsw**] [*name*...]

DESCRIPTION

Finger is used to find out about people. It searches for the local */etc/passwd* file and the Yellow Pages for matching account names and first or last names. *Finger* displays, if known, the name of the person associated with each account, mail alias, and home and office telephone numbers. If there is an account on the local machine, the home directory and login shell, and any plan which the person has placed in the file *.plan* in their home directory, and the project on which they are working from the file *.project* also in the home directory.

Finger may be used to lookup users on a remote machine by specfying their names using *user@host*.

If invoked without *name* arguments, *finger* lists information about all users currently logged in, using a short output format.

Finger options include:

–b	Force brief output format.
–f	Suppress printing of the *.project* files
–h	Suppress column headers if invoked without arguments.
–i	Display idle time using quick output format.
–l	Force long (default) output format.
–m	Match arguments only on user name.
–p	Suppress printing of the *.plan* files
–q	Force quick output format.
–s	Force short output format.
–w	Force narrow output format.

FILES

/etc/passwd	for users names, offices, ...
~/.plan	plans
~/.project	projects

SEE ALSO

passmgmt(1M), who(1), ypchpass(1)

BUGS

Only the first line of the *.project* file is printed.

There is no way to pass arguments to the remote machine as *finger* uses an internet standard port.

NAME

fmt – simple text formatter

SYNOPSIS

fmt [–width] [name ...]

DESCRIPTION

Fmt is a simple text formatter which reads the concatenation of input files (or standard input if none are given) and produces on standard output a version of its input with lines as close to *width* characters long as possible. The spacing at the beginning of the input lines is preserved in the output, as are blank lines and interword spacing. The default *width* is 72 characters.

Fmt is meant to format mail messages prior to sending, but may also be useful for other simple tasks. For instance, within visual mode of the *ex* editor (e.g. *vi*) the command

```
!}fmt
```

will reformat a paragraph, justifying the lines.

SEE ALSO

nroff(1), Mail(1)

BUGS

The program was designed to be simple and fast – for more complex operations, the standard text processors are likely to be more appropriate.

NAME

fold – fold long lines for finite width output device

SYNOPSIS

fold [−width] [file ...]

DESCRIPTION

Fold is a filter which will fold the contents of the specified files, or the standard input if no files are specified, breaking the lines to have maximum width *width.* The default for *width* is 80. *Width* should be a multiple of 8 if tabs are present.

BUGS

If underlining is present it may be messed up by folding.

NAME

ftimer – control clock and itimer resolution

SYNOPSIS

ftimer [-f on|off] [-i on|off]

DESCRIPTION

ftimer provides control of clock and itimer resolution. By default, the resolution for *gettimeofday(2)* and *setitimer(2)* is 100 HZ (which equals 10 milliseconds).

Higher resolution of *gettimeofday(2)* is enabled and disabled via the *-f* flag. The resolution when *on* is 1200 HZ, though this may be modified by changing the definition of FASTHZ in the file master.d/kernel(see lboot(1M)). The alternate value of FASTHZ however can only be either 900 Hz or 1800 Hz. The kernel will use the default value of 1200 HZ, if any other value is given. If the variable fastclock is set in the file master.d/kernel(see lboot(1M)) then the system fast clock is automatically enabled at boot time. *ftimer* only enables the fast clock on the CPU that is currently handling the clock function. Therefore on multi-processor systems, which CPU controls the fast clock may be selected using the *mpadmin*(1) command.

Higher resolution real itimers are enabled and disabled via the *-i* flag. The resolution of the real itimers when *on* is the same as for the fast clock, when *off* it is 100 HZ. Note that the overhead for a real itimer request that requires the additional resolution will put an approximate 6-8% performance penalty on the system.

Given no arguments, *ftimer* prints the status of the fast clock and fast itimers.

Only super user can turn on the fast clock.

SEE ALSO

getitimer(2), lboot(1), mpadmin(1).

NAME

ftoc – interface between prof and cord

SYNOPSIS

ftoc file1 ...

DESCRIPTION

ftoc reads one or more feedback files produced by the **–feedback** option of the profiler *prof*(1) and writes onto stdout a reorder-file for use with the procedure-rearranging program *cord*(1). It interprets each feedback file as representing one phase of a program's execution. In other words, if a program behaves in two distinct ways depending on its input, you could create two different feedback files by executing the program twice with different input data, and both *ftoc* and *cord* will understand that the information from the first file is distinct from that of the second file.

As an example, to improve the instruction-cache performance of a program called *hello,* you could generate a new *hello.cord* program by issuing the commands:

```
cc -o hello hello.c -Wl,-r,-d,-z
pixie -o hello.pix hello
hello.pix
prof -pixie -feedback hello.fd hello
/usr/lib/ftoc hello.fd > hello.reorder
/usr/lib/cord -o hello.cord hello hello.reorder
```

The reorderfile consists of a list of lines of the form:

```
        sourcefile procname.procname... n
```

where "procname.procname..." represents an outer-to-inner list of nested procedures, and *n* is 10 times the percentage of the procedure's "density" with respect to the total of the densities of all procedures. ("Density" is the ratio of a procedure's total cycles to its total static instructions.) A line consisting of "$phase" separates information from different feedback files.

FILES

/usr/lib/ftoc

SEE ALSO

cord(1), prof(1)

NAME

ftp – Internet file transfer program

SYNOPSIS

ftp [**–v**] [**–d**] [**–i**] [**–n**] [**–g**] [**host**]

DESCRIPTION

Ftp is the user interface to the Internet standard File Transfer Protocol. The program allows a user to transfer files to and from a remote network site.

The client host with which *ftp* is to communicate may be specified on the command line. If this is done, *ftp* will immediately attempt to establish a connection to an FTP server on that host; otherwise, *ftp* will enter its command interpreter and await instructions from the user. When *ftp* is awaiting commands from the user the prompt ''ftp>'' is provided to the user. The following commands are recognized by *ftp*:

! [*command* [*args*]]
: Invoke an interactive shell on the local machine. If there are arguments, the first is taken to be a command to execute directly, with the rest of the arguments as its arguments.

$ *macro-name* [*args*]
: Execute the macro *macro-name* that was defined with the **macdef** command. Arguments are passed to the macro unglobbed.

account [*passwd*]
: Supply a supplemental password required by a remote system for access to resources once a login has been successfully completed. If no argument is included, the user will be prompted for an account password in a non-echoing input mode.

append *local-file* [*remote-file*]
: Append a local file to a file on the remote machine. If *remote-file* is left unspecified, the local file name is used in naming the remote file after being altered by any *ntrans* or *nmap* setting. File transfer uses the current settings for *type*, *format*, *mode*, and *structure*.

ascii Set the file transfer *type* to network ASCII. This is the default type.

bell Arrange that a bell be sounded after each file transfer command is completed.

binary Set the file transfer *type* to support binary image transfer.

bye Terminate the FTP session with the remote server and exit *ftp*. An end of file will also terminate the session and exit.

case Toggle remote computer file name case mapping during **mget** commands. When **case** is on (default is off), remote computer file names with all letters in upper case are written in the local directory with the letters mapped to lower case.

cd *remote-directory*
Change the working directory on the remote machine to *remote-directory*.

cdup Change the remote machine working directory to the parent of the current remote machine working directory.

chmod *mode file-name*
Change the permission modes for the file *file-name* on the remote sytem to *mode*.

close Terminate the FTP session with the remote server, and return to the command interpreter. Any defined macros are erased.

cr Toggle carriage return stripping during ascii type file retrieval. Records are denoted by a carriage return/linefeed sequence during ascii type file transfer. When **cr** is on (the default), carriage returns are stripped from this sequence to conform with the UNIX single linefeed record delimiter. Records on non-UNIX remote systems may contain single linefeeds; when an ascii type transfer is made, these linefeeds may be distinguished from a record delimiter only when **cr** is off.

delete *remote-file*
Delete the file *remote-file* on the remote machine.

debug [*debug-value*]
Toggle debugging mode. If an optional *debug-value* is specified it is used to set the debugging level. When debugging is on, *ftp* prints each command sent to the remote machine, preceded by the string "-->".

dir [*remote-directory*] [*local-file*]
Print a listing of the directory contents in the directory, *remote-directory*, and, optionally, placing the output in *local-file*. If interactive prompting is on, *ftp* will prompt the user to verify that the last argument is indeed the target local file for receiving **dir** output. If no directory is specified, the current working directory on the remote machine is used. If no local file is specified, or *local-file* is –, output comes to the terminal.

disconnect
A synonym for **close**.

form *format*
Set the file transfer *form* to *format*. The default format is "file".

get *remote-file* [*local-file*]
Retrieve the *remote-file* and store it on the local machine. If the local file name is not specified, it is given the same name it has on the remote machine, subject to alteration by the current *case*, *ntrans*, and *nmap* settings. The current settings for *type*, *form*, *mode*, and *structure* are used while transferring the file.

glob
Toggle filename expansion for **mdelete, mget** and **mput**. If globbing is turned off with **glob**, the file name arguments are taken literally and not expanded. Globbing for **mput** is done as in **csh**(1). For **mdelete** and **mget**, each remote file name is expanded separately on the remote machine and the lists are not merged. Expansion of a directory name is likely to be different from expansion of the name of an ordinary file: the exact result depends on the foreign operating system and ftp server, and can be previewed by doing '**mls** *remote-files* –'. Note: **mget** and **mput** are not meant to transfer entire directory subtrees of files. That can be done by transferring a **tar**(1) archive of the subtree (in binary mode).

hash
Toggle hash-sign ("#") printing for each data block transferred. The size of a data block is 1024 bytes.

help [*command*]
Print an informative message about the meaning of *command*. If no argument is given, *ftp* prints a list of the known commands.

idle [*seconds*]
Set the inactivity timer on the remote server to *seconds* seconds. If *seconds* is omitted, the current inactivity timer is printed.

lcd [*directory*]
Change the working directory on the local machine. If no *directory* is specified, the user's home directory is used.

ls [*remote-directory*] [*local-file*]
Print a listing of the contents of a directory on the remote machine. The listing includes any system-dependent information that the server chooses to include; for example, most UNIX systems will produce output from the command "ls –lA". (See also **nlist.**) If *remote-directory* is left unspecified, the current working directory is used. If interactive prompting is on, *ftp* will prompt the user to

verify that the last argument is indeed the target local file for receiving **ls** output. If no local file is specified, or if *local-file* is –, the output is sent to the terminal.

macdef *macro-name*
Define a macro. Subsequent lines are stored as the macro *macro-name*; a null line (consecutive newline characters in a file or carriage returns from the terminal) terminates macro input mode. There is a limit of 16 macros and 4096 total characters in all defined macros. Macros remain defined until a **close** command is executed. The macro processor interprets '$' and '\' as special characters. A '$' followed by a number (or numbers) is replaced by the corresponding argument on the macro invocation command line. A '$' followed by an 'i' signals that macro processor that the executing macro is to be looped. On the first pass '$i' is replaced by the first argument on the macro invocation command line, on the second pass it is replaced by the second argument, and so on. A '\' followed by any character is replaced by that character. Use the '\' to prevent special treatment of the '$'.

mdelete [*remote-files*]
Delete the *remote-files* on the remote machine.

mdir *remote-files local-file*
Like **dir**, except multiple remote files may be specified. If interactive prompting is on, *ftp* will prompt the user to verify that the last argument is indeed the target local file for receiving **mdir** output.

mget *remote-files*
Expand the *remote-files* on the remote machine and do a **get** for each file name thus produced. See **glob** for details on the filename expansion. Resulting file names will then be processed according to *case*, *ntrans*, and *nmap* settings. Files are transferred into the local working directory, which can be changed with '**lcd** directory'; new local directories can be created with '**!** mkdir directory'.

mkdir *directory-name*
Make a directory on the remote machine.

mls *remote-files local-file*
Like **nlist**, except multiple remote files may be specified, and the *local-file* must be specified. If interactive prompting is on, *ftp* will prompt the user to verify that the last argument is indeed the target local file for receiving **mls** output.

mode [*mode-name*]
: Set the file transfer *mode* to *mode-name*. The default mode is ``stream'' mode.

modtime *file-name*
: Show the last modification time of the file on the remote machine.

mput *local-files*
: Expand wild cards in the list of local files given as arguments and do a **put** for each file in the resulting list. See **glob** for details of filename expansion. Resulting file names will then be processed according to *ntrans* and *nmap* settings.

newer *file-name*
: Get the file only if the modification time of the remote file is more recent that the file on the current system. If the file does not exist on the current system, the remote file is considered *newer*. Otherwise, this command is identical to **get**.

nlist [*remote-directory*] [*local-file*]
: Print a list of the files of a directory on the remote machine. If *remote-directory* is left unspecified, the current working directory is used. If interactive prompting is on, *ftp* will prompt the user to verify that the last argument is indeed the target local file for receiving **nlist** output. If no local file is specified, or if *local-file* is –, the output is sent to the terminal.

nmap [*inpattern outpattern*]
: Set or unset the filename mapping mechanism. If no arguments are specified, the filename mapping mechanism is unset. If arguments are specified, remote filenames are mapped during **mput** commands and **put** commands issued without a specified remote target filename. If arguments are specified, local filenames are mapped during **mget** commands and **get** commands issued without a specified local target filename. This command is useful when connecting to a non-UNIX remote computer with different file naming conventions or practices. The mapping follows the pattern set by *inpattern* and *outpattern*. *Inpattern* is a template for incoming filenames (which may have already been processed according to the **ntrans** and **case** settings). Variable templating is accomplished by including the sequences '$1', '$2', ..., '$9' in *inpattern*. Use '\' to prevent this special treatment of the '$' character. All other characters are treated literally, and are used to determine the **nmap** *inpattern* variable values. For example, given *inpattern* $1.$2 and the remote file name "mydata.data", $1 would have the value "mydata", and $2 would have the value "data". The *outpattern* determines the resulting mapped filename. The sequences

'$1', '$2', ..., '$9' are replaced by any value resulting from the *inpattern* template. The sequence '$0' is replace by the original filename. Additionally, the sequence '[*seq1*,*seq2*]' is replaced by *seq1* if *seq1* is not a null string; otherwise it is replaced by *seq2*. For example, the command "nmap $1.$2.$3 [$1,$2].[$2,file]" would yield the output filename "myfile.data" for input filenames "myfile.data" and "myfile.data.old", "myfile.file" for the input filename "myfile", and "myfile.myfile" for the input filename ".myfile". Spaces may be included in *outpattern*, as in the example: nmap $1 |sed "s/ *$//" > $1 . Use the '\' character to prevent special treatment of the '$', '[', ']', and ',' characters.

ntrans [*inchars* [*outchars*]]
Set or unset the filename character translation mechanism. If no arguments are specified, the filename character translation mechanism is unset. If arguments are specified, characters in remote filenames are translated during **mput** commands and **put** commands issued without a specified remote target filename. If arguments are specified, characters in local filenames are translated during **mget** commands and **get** commands issued without a specified local target filename. This command is useful when connecting to a non-UNIX remote computer with different file naming conventions or practices. Characters in a filename matching a character in *inchars* are replaced with the corresponding character in *outchars*. If the character's position in *inchars* is longer than the length of *outchars*, the character is deleted from the file name.

open *host* [*port*]
Establish a connection to the specified *host* FTP server. An optional port number may be supplied, in which case, *ftp* will attempt to contact an FTP server at that port. If the *auto-login* option is on (default), *ftp* will also attempt to automatically log the user in to the FTP server (see below).

prompt Toggle interactive prompting. Interactive prompting occurs during multiple file transfers to allow the user to selectively retrieve or store files. If prompting is turned off (default is on), any **mget** or **mput** will transfer all files, and any **mdelete** will delete all files.

proxy *ftp-command*
Execute an ftp command on a secondary control connection. This command allows simultaneous connection to two remote ftp servers for transferring files between the two servers. The first **proxy** command should be an **open**, to establish the secondary control connection. Enter the command "proxy ?" to see other ftp

commands executable on the secondary connection. The following commands behave differently when prefaced by **proxy**: **open** will not define new macros during the auto-login process, **close** will not erase existing macro definitions, **get** and **mget** transfer files from the host on the primary control connection to the host on the secondary control connection, and **put**, **mput**, and **append** transfer files from the host on the secondary control connection to the host on the primary control connection. Third party file transfers depend upon support of the ftp protocol PASV command by the server on the secondary control connection.

put *local-file* [*remote-file*]
: Store a local file on the remote machine. If *remote-file* is left unspecified, the local file name is used after processing according to any *ntrans* or *nmap* settings in naming the remote file. File transfer uses the current settings for *type*, *format*, *mode*, and *structure*.

pwd Print the name of the current working directory on the remote machine.

quit A synonym for **bye**.

quote *arg1 arg2 ...*
: The arguments specified are sent, verbatim, to the remote FTP server.

recv *remote-file* [*local-file*]
: A synonym for get.

reget *remote-file* [*local-file*]
: Reget acts like get, except that if *local-file* exists and is smaller than *remote-file*, *local-file* is presumed to be a partially transferred copy of *remote-file* and the transfer is continued from the apparent point of failure. This command is useful when transferring very large files over networks that are prone to dropping connections.

remotehelp [*command-name*]
: Request help from the remote FTP server. If a *command-name* is specified it is supplied to the server as well.

remotestatus [*file-name*]
: With no arguments, show status of remote machine. If *file-name* is specified, show status of *file-name* on remote machine.

rename [*from*] [*to*]
: Rename the file *from* on the remote machine, to the file *to*.

reset Clear reply queue. This command re-synchronizes command/reply sequencing with the remote ftp server. Resynchronization may be necessary following a violation of the ftp protocol by the remote server.

restart *marker*
Restart the immediately following **get** or **put** at the indicated *marker*. On UNIX systems, marker is usually a byte offset into the file.

rmdir *directory-name*
Delete a directory on the remote machine.

runique
Toggle storing of files on the local system with unique filenames. If a file already exists with a name equal to the target local filename for a **get** or **mget** command, a ".1" is appended to the name. If the resulting name matches another existing file, a ".2" is appended to the original name. If this process continues up to ".99", an error message is printed, and the transfer does not take place. The generated unique filename will be reported. Note that **runique** will not affect local files generated from a shell command (see below). The default value is off.

send *local-file* [*remote-file*]
A synonym for put.

sendport
Toggle the use of PORT commands. By default, *ftp* will attempt to use a PORT command when establishing a connection for each data transfer. The use of PORT commands can prevent delays when performing multiple file transfers. If the PORT command fails, *ftp* will use the default data port. When the use of PORT commands is disabled, no attempt will be made to use PORT commands for each data transfer. This is useful for certain FTP implementations which do ignore PORT commands but, incorrectly, indicate they've been accepted.

site *arg1 arg2 ...*
The arguments specified are sent, verbatim, to the remote FTP server as a SITE command.

size *file-name*
Return size of *file-name* on remote machine.

status Show the current status of *ftp*.

struct [*struct-name*]
Set the file transfer *structure* to *struct-name*. By default ''stream'' structure is used.

sunique
Toggle storing of files on remote machine under unique file names. Remote ftp server must support ftp protocol STOU command for successful completion. The remote server will report unique name. Default value is off.

system Show the type of operating system running on the remote machine.

tenex Set the file transfer type to that needed to talk to TENEX machines.

trace Toggle packet tracing.

type [*type-name*]
Set the file transfer *type* to *type-name*. If no type is specified, the current type is printed. The default type is network ASCII.

umask [*newmask*]
Set the default umask on the remote server to *newmask*. If *newmask* is omitted, the current umask is printed.

user *user-name* [*password*] [*account*]
Identify yourself to the remote FTP server. If the password is not specified and the server requires it, *ftp* will prompt the user for it (after disabling local echo). If an account field is not specified, and the FTP server requires it, the user will be prompted for it. If an account field is specified, an account command will be relayed to the remote server after the login sequence is completed if the remote server did not require it for logging in. Unless *ftp* is invoked with ''auto-login'' disabled, this process is done automatically on initial connection to the FTP server.

verbose Toggle verbose mode. In verbose mode, all responses from the FTP server are displayed to the user. In addition, if verbose is on, when a file transfer completes, statistics regarding the efficiency of the transfer are reported. By default, verbose is on.

? [*command*]
A synonym for help.

Command arguments which have embedded spaces may be quoted with quote (") marks.

ABORTING A FILE TRANSFER

To abort a file transfer, use the terminal interrupt key (usually Ctrl-C). Sending transfers will be immediately halted. Receiving transfers will be halted by sending a ftp protocol ABOR command to the remote server, and discarding any further data received. The speed at which this is accomplished depends upon the remote server's support for ABOR processing. If the remote server does not support the ABOR command, an "ftp>" prompt will not appear until the remote server has completed sending the requested file.

The terminal interrupt key sequence will be ignored when *ftp* has completed any local processing and is awaiting a reply from the remote server. A long delay in this mode may result from the ABOR processing described above, or from unexpected behavior by the remote server, including violations of the ftp protocol. If the delay results from unexpected remote server behavior, the local *ftp* program must be killed by hand.

FILE NAMING CONVENTIONS

Files specified as arguments to *ftp* commands are processed according to the following rules.

1) If the file name ‘‘–’’ is specified, the **stdin** (for reading) or **stdout** (for writing) is used.

2) If the first character of the file name is ‘‘|’’, the remainder of the argument is interpreted as a shell command. *Ftp* then forks a shell, using *popen*(3) with the argument supplied, and reads (writes) from the stdout (stdin). If the shell command includes spaces, the argument must be quoted; e.g., ‘‘"| ls –lt"’’. A particularly useful example of this mechanism is: ‘‘dir |more’’.

3) Failing the above checks, if ‘‘globbing’’ is enabled, local file names are expanded according to the rules used in the *csh*(1); c.f. the *glob* command. If the *ftp* command expects a single local file (e.g., **put**), only the first filename generated by the "globbing" operation is used.

4) For **mget** commands and **get** commands with unspecified local file names, the local filename is the remote filename, which may be altered by a **case**, **ntrans**, or **nmap** setting. The resulting filename may then be altered if **runique** is on.

5) For **mput** commands and **put** commands with unspecified remote file names, the remote filename is the local filename, which may be altered by a **ntrans** or **nmap** setting. The resulting filename may then be altered by the remote server if **sunique** is on.

FILE TRANSFER PARAMETERS

The FTP specification specifies many parameters which may affect a file transfer. The *type* may be one of "ascii", "image" (binary), "ebcdic", and "local byte size" (for PDP-10's and PDP-20's mostly). *Ftp* supports the ascii and image types of file transfer, plus local byte size 8 for **tenex** mode transfers.

Ftp supports only the default values for the remaining file transfer parameters: *mode*, *form*, and *struct*.

OPTIONS

Options may be specified at the shell command line. Several options can be enabled or disabled with *ftp* commands.

The **–v** (verbose on) option forces *ftp* to show all responses from the remote server, as well as report on data transfer statistics.

The **–n** option restrains *ftp* from attempting "auto-login" upon initial connection. If auto-login is enabled, *ftp* will check the *.netrc* file (see below) in the user's home directory for an entry describing an account on the remote machine. If no entry exists, *ftp* will prompt for the remote machine login name (default is the user identity on the local machine), and, if necessary, prompt for a password and an account with which to login.

The **–i** option turns off interactive prompting during multiple file transfers.

The **–d** option enables debugging.

The **–g** option disables file name globbing.

THE .netrc FILE

The .netrc file contains login and initialization information used by the auto-login process. It resides in the user's home directory. The following tokens are recognized; they may be separated by spaces, tabs, or new-lines:

machine *name*
: Identify a remote machine name. The auto-login process searches the .netrc file for a **machine** token that matches the remote machine specified on the *ftp* command line or as an **open** command argument. Once a match is made, the subsequent .netrc tokens are processed, stopping when the end of file is reached or another **machine** or a **default** token is encountered.

default
: This is the same as **machine** *name* except that **default** matches any name. There can be only one **default** token, and it must be after all **machine** tokens. This is normally used as:

    default login anonymous password user@site

  thereby giving the user *automatic* anonymous ftp login to machines not specified in **.netrc**. This can be overridden by using the **–n** flag to disable auto-login.

login *name*
: Identify a user on the remote machine. If this token is present, the auto-login process will initiate a login using the specified name.

password *string*
: Supply a password. If this token is present, the auto-login process will supply the specified string if the remote server requires a password as part of the login process. Note that if this token is present in the .netrc file for any user other than *anonymous*, *ftp* will abort the auto-login process if the .netrc is readable by anyone besides the user.

account *string*
: Supply an additional account password. If this token is present, the auto-login process will supply the specified string if the remote server requires an additional account password, or the auto-login process will initiate an ACCT command if it does not. Note that if this token is present in the .netrc file, *ftp* will abort the auto-login process if the .netrc is readable by anyone besides the user.

macdef *name*
: Define a macro. This token functions like the *ftp* **macdef** command functions. A macro is defined with the specified name; its contents begin with the next .netrc line and continue until a null line (consecutive new-line characters) is encountered. If a macro named *init* is defined, it is automatically executed as the last step in the auto-login process.

SEE ALSO

ftpd(1M)

BUGS

Correct execution of many commands depends upon proper behavior by the remote server.

An error in the treatment of carriage returns in the 4.2BSD UNIX ascii-mode transfer code has been corrected. This correction may result in incorrect transfers of binary files to and from 4.2BSD servers using the ascii type. Avoid this problem by using the binary image type.

G – H

NAME

gclear – clear IRIS graphics screen

SYNOPSIS

gclear

DESCRIPTION

Gclear clears the entire IRIS graphics screen.

NAME

get – get a version of an SCCS file

SYNOPSIS

get [**–r**SID] [**–c**cutoff] [**–i**list] [**–x**list] [**–w**string] [**–a**seq-no.] [**–k**] [**–e**] [**–l**[**p**] [**–p**] [**–m**] [**–n**] [**–s**] [**–b**] [**–g**] [**–t**] file ...

DESCRIPTION

get generates an ASCII text file from each named SCCS file according to the specifications given by its keyletter arguments, which begin with –. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *get* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The generated text is normally written into a file called the *g-file* whose name is derived from the SCCS file name by simply removing the leading **s.**; (see also *FILES*, below).

Each of the keyletter arguments is explained below as though only one SCCS file is to be processed, but the effects of any keyletter argument applies independently to each named file.

–r*SID* The *S*CCS *ID*entification string (SID) of the version (delta) of an SCCS file to be retrieved. Table˜1 below shows, for the most useful cases, what version of an SCCS file is retrieved (as well as the SID of the version to be eventually created by *delta*(1) if the **–e** keyletter is also used), as a function of the SID specified.

–c*cutoff* *Cutoff* date-time, in the form:

YY[MM[DD[HH[MM[SS]]]]]

No changes (deltas) to the SCCS file which were created after the specified *cutoff* date-time are included in the generated ASCII text file. Units omitted from the date-time default to their maximum possible values; that is, **–c7502** is equivalent to **–c750228235959**. Any number of non-numeric characters may separate the various 2-digit pieces of the *cutoff* date-time. This feature allows one to specify a *cutoff* date in the form: "**–c77/2/2 9:22:25**". Note that this implies that one may use the %E% and %U% identification keywords (see below) for nested *gets* within, say the input to a *send*(1C) command:

```
~!get "–c%E% %U%" s.file
```

–i*list* A *list* of deltas to be included (forced to be applied) in the creation of the generated file. The *list* has the following syntax:

```
<list> ::= <range> | <list> , <range>
<range> ::= SID | SID – SID
```

SID, the SCCS Identification of a delta, may be in any form shown in the "SID Specified" column of Table 1.

–x*list* A *list* of deltas to be excluded in the creation of the generated file. See the **–i** keyletter for the *list* format.

–e Indicates that the *get* is for the purpose of editing or making a change (delta) to the SCCS file via a subsequent use of *delta*(1). The **–e** keyletter used in a *get* for a particular version (SID) of the SCCS file prevents further *get*s for editing on the same SID until *delta* is executed or the **j** (joint edit) flag is set in the SCCS file [see *admin*(1)]. Concurrent use of get **–e** for different SIDs is always allowed.

If the *g-file* generated by *get* with an **–e** keyletter is accidentally ruined in the process of editing it, it may be regenerated by re-executing the *get* command with the **–k** keyletter in place of the **–e** keyletter.

SCCS file protection specified via the ceiling, floor, and authorized user list stored in the SCCS file [see *admin*(1)] are enforced when the **–e** keyletter is used.

–b Used with the **–e** keyletter to indicate that the new delta should have an SID in a new branch as shown in Table 1. This keyletter is ignored if the b flag is not present in the file [see *admin*(1)] or if the retrieved *delta* is not a leaf *delta*. (A leaf *delta* is one that has no successors on the SCCS file tree.)
Note: A branch *delta* may always be created from a non-leaf *delta*. Partial SIDs are interpreted as shown in the "SID Retrieved" column of Table 1.

–k Suppresses replacement of identification keywords (see below) in the retrieved text by their value. The **–k** keyletter is implied by the **–e** keyletter.

−l[p] Causes a delta summary to be written into an *l-file*. If **−lp** is used then an *l-file* is not created; the delta summary is written on the standard output instead. See *FILES* for the format of the *l-file*.

−p Causes the text retrieved from the SCCS file to be written on the standard output. No *g-file* is created. All output which normally goes to the standard output goes to file descriptor 2 instead, unless the **−s** keyletter is used, in which case it disappears.

−s Suppresses all output normally written on the standard output. However, fatal error messages (which always go to file descriptor 2) remain unaffected.

−m Causes each text line retrieved from the SCCS file to be preceded by the SID of the delta that inserted the text line in the SCCS file. The format is: SID, followed by a horizontal tab, followed by the text line.

−n Causes each generated text line to be preceded with the %M% identification keyword value (see below). The format is: %M% value, followed by a horizontal tab, followed by the text line. When both the **−m** and **−n** keyletters are used, the format is: %M% value, followed by a horizontal tab, followed by the **−m** keyletter generated format.

−g Suppresses the actual retrieval of text from the SCCS file. It is primarily used to generate an *l-file*, or to verify the existence of a particular SID.

−t Used to access the most recently created delta in a given release (e.g., **−r1**), or release and level (e.g., **−r1.2**).

−w *string* Substitute *string* for all occurrences of %W% when getting the file.

−a*seq-no.* The delta sequence number of the SCCS file delta (version) to be retrieved [see *sccsfile*(5)]. This keyletter is used by the *comb*(1) command; it is not a generally useful keyletter. If both the **−r** and **−a** keyletters are specified, only the **−a** keyletter is used. Care should be taken when using the **−a** keyletter in conjunction with the **−e** keyletter, as the SID of the delta to be created may not be what one expects. The **−r** keyletter can be used with the **−a** and **−e** keyletters to control the naming of the SID of the delta to be created.

For each file processed, *get* responds (on the standard output) with the SID being accessed and with the number of lines retrieved from the SCCS file.

If the **–e** keyletter is used, the SID of the delta to be made appears after the SID accessed and before the number of lines generated. If there is more than one named file or if a directory or standard input is named, each file name is printed (preceded by a new-line) before it is processed. If the **–i** keyletter is used included deltas are listed following the notation ''Included''; if the **–x** keyletter is used, excluded deltas are listed following the notation ''Excluded''.

TABLE 1. Determination of SCCS Identification String

SID* Specified	–b Keyletter Used†	Other Conditions	SID Retrieved	SID of Delta to be Created
none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1).1
R	no	R > mR	mR.mL	R.1***
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
R	–	R < mR and R does *not* exist	hR.mL**	hR.mL.(mB+1).1
R	–	Trunk succ.# in release > R and R exists	R.mL	R.mL.(mB+1).1
R.L	no	No trunk succ.	R.L	R.(L+1)
R.L	yes	No trunk succ.	R.L	R.L.(mB+1).1
R.L	–	Trunk succ. in release ≥ R	R.L	R.L.(mB+1).1
R.L.B	no	No branch succ.	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch succ.	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch succ.	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch succ.	R.L.B.S	R.L.(mB+1).1
R.L.B.S	–	Branch succ.	R.L.B.S	R.L.(mB+1).1

*	"R", "L", "B", and "S" are the "release", "level", "branch", and "sequence" components of the SID, respectively; "m" means "maximum". Thus, for example, "R.mL" means "the maximum level number within release R"; "R.L.(mB+1).1" means "the first sequence number on the *new* branch (i.e., maximum branch number plus one) of level L within release R". Note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components *must* exist.
**	"hR" is the highest *existing* release that is lower than the specified, *nonexistent*, release R.
***	This is used to force creation of the *first* delta in a *new* release.
#	Successor.
†	The **−b** keyletter is effective only if the **b** flag [see *admin*(1)] is present in the file. An entry of – means "irrelevant".
‡	This case applies if the **d** (default SID) flag is *not* present in the file. If the **d** flag *is* present in the file, then the SID obtained from the **d** flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

IDENTIFICATION KEYWORDS

Identifying information is inserted into the text retrieved from the SCCS file by replacing *identification keywords* with their value wherever they occur. The following keywords may be used in the text stored in an SCCS file:

Keyword	*Value*
%M%	Module name: either the value of the **m** flag in the file [see *admin*(1)], or if absent, the name of the SCCS file with the leading s. removed.
%I%	SCCS identification (SID) (%R%.%L%.%B%.%S%) of the retrieved text.
%R%	Release.
%L%	Level.
%B%	Branch.
%S%	Sequence.
%D%	Current date (YY/MM/DD).
%H%	Current date (MM/DD/YY).
%T%	Current time (HH:MM:SS).
%E%	Date newest applied delta was created (YY/MM/DD).
%G%	Date newest applied delta was created (MM/DD/YY).
%U%	Time newest applied delta was created (HH:MM:SS).
%Y%	Module type: value of the **t** flag in the SCCS file [see *admin*(1)].
%F%	SCCS file name.

%P%	Fully qualified SCCS file name.
%Q%	The value of the **q** flag in the file [see *admin*(1)].
%C%	Current line number. This keyword is intended for identifying messages output by the program such as ``this should not have happened'' type errors. It is *not* intended to be used on every line to provide sequence numbers.
%Z%	The 4-character string @(#) recognizable by *what*(1).
%W%	A shorthand notation for constructing *what*(1) strings for UNIX system program files. %W% = %Z%%M%<horizontal-tab>%I%
%A%	Another shorthand notation for constructing *what*(1) strings for non-UNIX system program files. %A% = %Z%%Y% %M% %I%%Z%

Several auxiliary files may be created by *get*. These files are known generically as the *g-file*, *l-file*, *p-file*, and *z-file*. The letter before the hyphen is called the tag. An auxiliary file name is formed from the SCCS file name: the last component of all SCCS file names must be of the form s.*module-name*, the auxiliary files are named by replacing the leading s with the tag. The *g-file* is an exception to this scheme: the *g-file* is named by removing the s. prefix. For example, **s.xyz.c**, the auxiliary file names would be **xyz.c**, **l.xyz.c**, **p.xyz.c**, and **z.xyz.c**, respectively.

The *g-file*, which contains the generated text, is created in the current directory (unless the **−p** keyletter is used). A *g-file* is created in all cases, whether or not any lines of text were generated by the *get*. It is owned by the real user. If the **−k** keyletter is used or implied its mode is 644; otherwise its mode is 444. Only the real user need have write permission in the current directory.

The *l-file* contains a table showing which deltas were applied in generating the retrieved text. The *l-file* is created in the current directory if the **−l** keyletter is used; its mode is 444 and it is owned by the real user. Only the real user need have write permission in the current directory.

Lines in the *l-file* have the following format:

a. A blank character if the delta was applied;
* otherwise.

b. A blank character if the delta was applied or was not applied and ignored;
* if the delta was not applied and was not ignored.

c. A code indicating a ``special'' reason why the delta was or was not applied:
``I'': Included.
``X'': Excluded.
``C'': Cut off (by a **−c** keyletter).

d. Blank.
e. SCCS identification (SID).
f. Tab character.
g. Date and time (in the form YY/MM/DD HH:MM:SS) of creation.
h. Blank.
i. Login name of person who created *delta*.

The comments and MR data follow on subsequent lines, indented one horizontal tab character. A blank line terminates each entry.

The *p-file* is used to pass information resulting from a *get* with an **−e** keyletter along to *delta*. Its contents are also used to prevent a subsequent execution of *get* with an **−e** keyletter for the same SID until *delta* is executed or the joint edit flag, **j**, [see *admin*(1)] is set in the SCCS file. The *p-file* is created in the directory containing the SCCS file and the effective user must have write permission in that directory. Its mode is 644 and it is owned by the effective user. The format of the *p-file* is: the gotten SID, followed by a blank, followed by the SID that the new delta will have when it is made, followed by a blank, followed by the login name of the real user, followed by a blank, followed by the date-time the *get* was executed, followed by a blank and the **−i** keyletter argument if it was present, followed by a blank and the **−x** keyletter argument if it was present, followed by a new-line. There can be an arbitrary number of lines in the *p-file* at any time; no two lines can have the same new delta SID.

The *z-file* serves as a *lock-out* mechanism against simultaneous updates. Its contents are the binary (2 bytes) process ID of the command (i.e., *get*) that created it. The *z-file* is created in the directory containing the SCCS file for the duration of *get*. The same protection restrictions as those for the *p-file* apply for the *z-file*. The *z-file* is created mode 444.

FILES

g-file	Existed before the execution of *delta*; removed after completion of *delta*.
p-file	Existed before the execution of *delta*; may exist after completion of *delta*.
q-file	Created during the execution of *delta*; removed after completion of *delta*.
x-file	Created during the execution of *delta*; renamed to SCCS file after completion of *delta*.
z-file	Created during the execution of *delta*; removed during the execution of *delta*.

d-file	Created during the execution of *delta*; removed after completion of *delta*.
/usr/bin/bdiff	Program to compute differences between the ''gotten'' file and the *g-file*.

SEE ALSO

admin(1), delta(1), prs(1), what(1).
help(1) in the *User's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

If the effective user has write permission (either explicitly or implicitly) in the directory containing the SCCS files, but the real user does not, then only one file may be named when the –e keyletter is used.

NAME

getopt – parse command options

SYNOPSIS

set -- ‘getopt optstring **$*‘**

DESCRIPTION

WARNING: Start using the new command *getopts*(1) in place of *getopt*(1). *getopt*(1) may not be supported in a future release. For more information, see the **WARNINGS** section below.

getopt is used to break up options in command lines for easy parsing by shell procedures and to check for legal options. *optstring* is a string of recognized option letters (see *getopt*(3C)); if a letter is followed by a colon, the option is expected to have an argument which may or may not be separated from it by white space. The special option **--** is used to delimit the end of the options. If it is used explicitly, *getopt* will recognize it; otherwise, *getopt* will generate it; in either case, *getopt* will place it at the end of the options. The positional parameters ($1 $2 ...) of the shell are reset so that each option is preceded by a – and is in its own positional parameter; each option argument is also parsed into its own positional parameter.

EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the options **a** or **b**, as well as the option **o**, which requires an argument:

```
set  --  `getopt  abo:  $*`
if  [  $?  !=  0  ]
then
        echo  $USAGE
        exit  2
fi
for  i  in  $*
do
        case  $i  in
        -a | -b)      FLAG=$i;  shift;;
        -o)           OARG=$2;  shift  2;;
        --)           shift;  break;;
        esac
done
```

This code will accept any of the following as equivalent:

```
cmd  -aoarg  file  file
cmd  -a  -o  arg  file  file
cmd  -oarg  -a  file  file
cmd  -a  -oarg  --  file  file
```

SEE ALSO

getopts(1), sh(1).
getopt(3C) in the *Programmer's Reference Manual.*

DIAGNOSTICS

getopt prints an error message on the standard error when it encounters an option letter not included in *optstring*.

WARNINGS

getopt(1) does not support the part of Rule 8 of the command syntax standard (see *intro*(1)) that permits groups of option-arguments following an option to be separated by white space and quoted. For example,

```
cmd -a -b -o "xxx z yy" file
```

is not handled correctly). To correct this deficiency, use the new command *getopts*(1) in place of *getopt*(1).

getopt(1) may not be supported in a future release. For this release a conversion tool has been provided, *getoptcvt*. For more information about *getopts* and *getoptcvt*, see the *getopts*(1) manual page.

If an option that takes an option-argument is followed by a value that is the same as one of the options listed in *optstring* (referring to the earlier EXAMPLE section, but using the following command line: `cmd -o -a file`), *getopt* will always treat **−a** as an option-argument to **−o**; it will never recognize **−a** as an option. For this case, the **for** loop in the example will shift past the *file* argument.

NAME

getopts, getoptcvt – parse command options

SYNOPSIS

getopts optstring name [arg ...]

/usr/lib/getoptcvt [**–b**] file

DESCRIPTION

getopts is a built-in command to *sh*(1) used to parse positional parameters and to check for legal options. It supports all applicable rules of the command syntax standard (see Rules 3-10, *intro*(1)). It should be used in place of the *getopt*(1) command. See the **WARNINGS** section below.

optstring must contain the option letters the command using *getopts* will recognize; if a letter is followed by a colon, the option is expected to have an argument, or group of arguments, which must be separated from it by white space.

Each time it is invoked, *getopts* will place the next option in the shell variable *name* and the index of the next argument to be processed in the shell variable **OPTIND**. Whenever the shell or a shell procedure is invoked, **OPTIND** is initialized to 1.

When an option requires an option-argument, *getopts* places it in the shell variable **OPTARG**.

If an illegal option is encountered, ? will be placed in *name*.

When the end of options is encountered, *getopts* exits with a non-zero exit status. The special option "--" may be used to delimit the end of the options.

By default, *getopts* parses the positional parameters. If extra arguments (*arg* ...) are given on the *getopts* command line, *getopts* will parse them instead.

/usr/lib/getoptcvt reads the shell script in *file*, converts it to use *getopts*(1) instead of *getopt*(1), and writes the results on the standard output.

–b the results of running */usr/lib/getoptcvt* will be portable to earlier releases of the UNIX system. */usr/lib/getoptcvt* modifies the shell script in *file* so that when the resulting shell script is executed, it determines at run time whether to invoke *getopts*(1) or *getopt*(1).

So all new commands will adhere to the command syntax standard described in *intro*(1), they should use *getopts*(1) or *getopt*(3C) to parse positional parameters and check for options that are legal for that command (see the **WARNINGS** section below).

EXAMPLE

The following fragment of a shell program shows how one might process the arguments for a command that can take the options **a** or **b**, as well as the option **o**, which requires an option-argument:

```
while getopts abo: c
do
        case $c in
        a | b)      FLAG=$c;;
        o)          OARG=$OPTARG;;
        \?)         echo $USAGE
                    exit 2;;
        esac
done
shift `expr $OPTIND - 1`
```

This code will accept any of the following as equivalent:

```
cmd -a -b -o "xxx z yy" file
cmd -a -b -o "xxx z yy" -- file
cmd -ab -o xxx,z,yy file
cmd -ab -o "xxx z yy" file
cmd -o xxx,z,yy -b -a file
```

SEE ALSO

intro(1), sh(1).
getopt(3C) in the *Programmer's Reference Manual*.

WARNINGS

Although the following command syntax rule (see *intro*(1)) relaxations are permitted under the current implementation, they should not be used because they may not be supported in future releases of the system. As in the **EXAMPLE** section above, **a** and **b** are options, and the option **o** requires an option-argument:

`cmd -aboxxx file` (Rule 5 violation: options with option-arguments must not be grouped with other options)
`cmd -ab -oxxx file` (Rule 6 violation: there must be white space after an option that takes an option-argument)

Changing the value of the shell variable **OPTIND** or parsing different sets of arguments may lead to unexpected results.

DIAGNOSTICS

getopts prints an error message on the standard error when it encounters an option letter not included in *optstring*.

NAME

gr_osview – graphical system monitor

SYNOPSIS

gr_osview **[-hVLapz]** [-**D**file] [-**N**[user@]node]

DESCRIPTION

This command provides a graphical display of usage of certain types of system resources. This display provides a real-time window into the overall operation of the system. A single rectangular moving bar consisting of bands of a particular color standing for each of the measured quantities is the main display type. The colors used, as well as size, layout, background and foreground colors may be modified. Each bar in a window has a header which consists of the bar title plus the names of each part of the resource displayed, in colors to match those used for each band in the bar.

The behavior described in this manual page is that of the second generation version of *gr_osview*. This version is not compatible with the original version at all. It is driven by a setup file. The setup file may be explicitly named on the command line, found through an environment variable, or in the caller's home directory. A complete description of the format and additional capabilities of the setup file is given below. *Gr_osview* is incompatible with previous versions for remote use; this means that a new version *gr_osview* can only talk to the same new version *gr_osview* on the remote system.

This man page describes version 2.1; this version is not generally compatible with previous versions.

OPTIONS

The following options are supported:

-h Print a summary of the available options.

-D file This option specifies where the setup file will come from. If the file "-" is named, then the setup will be read from standard input. If this option is not specified, the environment variable GROSVIEW is checked for a setup file name. If this variable is not set, then a file named ".grosview" is scanned for in the invoker's home directory. If all this fails, a default setup of a single CPU usage bar is used.

-N [user@]host
Contact the remote host, using the given user ID if specified, and monitor that host's performance on the local machine. The window border is forced in this case, and the host name (plus user name if specified) is used as the window title. This program is its own demon for remote use, and thus looks for a

program on the remote with the same path.

-V	Print the current version and copyright information for the program.
-a	This option is similar to the same option in old *gr_osview*, in that it invokes a subset of the displayable bars that matches the old incarnation.
-p	This option prints the window position and size after the window has been laid out on standard input. This is useful for programming setup scripts.
-z	This option enables "freeze" toggling. Sending SIGUSR1 (see *signal*(2)) to *gr_osview* will freeze and unfreeze the display. This is useful when doing screen dumps.
-L	This option causes *gr_osview* to lock down those pages which it actually uses while running and prevent the process from swapping. This enables a minimum number of pages to be locked while keeping *gr_osview* performing as a real-time display under heavy system load.

OVERVIEW

There are three potential formats for each bar, with various optional features for each. Each bar format includes a header line, which gives the bar name and the names of its parameters, plus some additional information depending on the bar type. Below this is a rectangular area in which is displayed information associated with each parameter, in either a graphical or textual manner. Parameters are such things as the various elements of CPU usage: user time, system time, interrupt overhead and idle time. The text of a parameter name in the header is colored to match the associated part of the bar display.

The most common format is called *moving bar format.* In this format, the information is displayed as a sectioned bar, each section being a different color. As time passes, each section will be redrawn with its length adjusted proportionally to the value of the parameter. To obtain smooth motion, and damp peaks and valleys which would otherwise be visually confusing, each section of the bar is allowed to grow or shrink only by a controlled amount. For instance, if a parameter doubles in value, the section will gradually double in length over a certain number of samples. By setting the sample interval, the user can control the update rate of the bar information. By default, a 2/10 second interval is used for smooth visual motion. A bar border is supplied by default which includes a heavy lower border marked in 1/10 increments for easier estimation. This border can be suppressed completely.

There are two subformats for the moving bar: relative and absolute. In relative mode, the displayed values must all add up to 100% and are relative only to each other. In absolute mode, the bar displays an indicator of events per second over the sample interval, and the header includes the sample interval. The bar itself will auto-scale; this means that the scale used will vary automatically (with some hysteresis) as the number of events changes. The scale value being used is displayed above the right-hand edge of the bar. The property of relative or absolute display is a characteristic of the type of bar being displayed, and is not under control of the user. The scale value can be *locked*, in which case autoscaling is disabled, and the scale value is inverted in color to indicate locked operation. The user may specify that the bar scale be locked, and what the bar scale should be in this case.

The second format is called *strip chart format*. Any moving bar may also be displayed as a strip chart. In this mode, instead of being displayed horizontally, the bar is drawn vertically at the right hand edge of the bar after moving the bar down by the size of a single sample. This forms a strip-chart effect. The number of samples and the sample interval can be changed. The header of an absolute bar displayed as a strip chart will indicate the overall time shown by the strip instead of the sample interval time. Finally, tick marks may be added to the strip to ease estimation.

If the strip chart format is used for an absolute bar type, and the scale is not locked, then autoscaling causes an additional action. When the scale changes, a red line is drawn through the bar at the point at which the scale changed, and the remainder of the bar is drawn in a gray color, showing only the outline of the parameter values. This shows that the scale changed, and the grayed out data is invalid.

In either of the above bar formats, and if the bar type is absolute, then some additional display options are available. In *max value mode*, the bar display area is compressed and a text field is added at the right hand of the bar showing the maximum value ever achieved by the sum of the parameters. By default, this number is displayed as red on black, in the upper part of the bar area. If *tracking mode* is enabled, then a text field is added at the right of the bar below the maximum value field showing the average of the sum of the parameters over an interval. By default, this number is displayed as blue on white. Each of these modes displays a calculated events-per-second ratio. If the sample interval is much smaller than one second, then the displays will show the "burst" rate achieved at the sample interval. The system may not be able to sustain this rate over longer periods of time. To get an accurate measure at any interval, simply adjust the sample interval as described below.

The third main format is *numeric format.* This format is currently only available for absolute value bar types. Instead of a graphic display, the bar is replaced by a text display area in which the actual values of the parameters are displayed. In contrast to the other displays, the parameter values are given over the whole sample interval rather than scaled to units per second. This allows a long sample time for slow changing parameters.

By default, the current values of the parameters are displayed in the same color as the graphical display would use just below each parameter name in the header. If *max value mode* is turned on, then a listing of the maximum value seen for each parameter is given below the current values, except that all text is displayed in the current maximum display foreground color, which is red by default. If *tracking mode* is turned on, then the average numeric values are summed and displayed to the right in the foreground color, with an appropriate title.

SETUP FILE

The setup file provides a simple mechanism for initializing a large number of possible parameters for the *gr_osview* display. The setup file is an ASCII file. Comment lines are delimited by a # character in the first column, and blank lines are ignored. In addition, trailing comments may be added using a # character, after which all data on the line is ignored. Lines containing information may be classed into two types, *monitor* lines and *option* lines. *Monitor* lines describe the format of an individual bar, while *option* lines describe global parameters. The monitor bars in the *gr_osview* window are brought up in the same order they are found in the setup file. A particular monitor bar may be entered several times, possibly with many different options.

Each monitor line consists of a name followed by zero or more modifier options. The following monitor bars are available:

cpu	- monitor CPU usage
rmem	- monitor real memory usage
rmemc	- monitor real memory usage
wait	- monitor time waiting for I/O
sysact	- monitor important system activity
gfx	- monitor important graphics activity
bdev	- monitor block device throughput
fault	- monitor page faults
tlb	- monitor TLB activity
intr	- monitor interrupts
disk	- monitor disk usage
pswap	- monitor page swapping activity
net	- monitor network activity

netif	- monitor network interface activity

The **cpu, netif** and **disk** bars have special formats. The **cpu** bar may have an optional argument which indicates a particular CPU to monitor (if more than one is present in the system). In this case, the descriptor takes the form:

cpu(n)	- monitor CPU number *n*

If the word *sum* is given instead, the bar monitors the sum of all CPU activity in the system in a single bar.

The **netif** bar may have an optional argument which indicates a particular interface to monitor, or it may indicate that only the sum of all network interface activity is to be monitored. By default, a bar is generated for each network interface on the system except for the local loopback interface. If a name is given, the descriptor takes the form:

netif(name)	- monitor interface name *name*

Typical interface names are *et0*, for the built-in ethernet interface on the POWERSeries, *lo0* for the local loopback interface, *enp0* for the standard Professional IRIS VMEbus ethernet card, and *ec0* for the Personal IRIS built-in ethernet interface. If the word *sum* is given instead, the bar monitors the sum of all interface traffic for the system.

The **disk** bar requires an argument describing the volume to monitor. The form of the descriptor is:

disk(path)	- monitor the volume given by *path*

The *path* argument can name a filesystem in one of two ways. If it names a block special device, then that device is assumed to contain an EFS filesystem, and usage is monitored with the special file name used as the bar header. Otherwise, the argument is assumed to name a file residing on some mounted volume. If the volume can be identified through the */etc/mnttab* file, then the device as named in the file is used as the bar header. Otherwise, the path name passed in is used as the header. The bar scale is set at the number of megabytes of storage on the volume.

The scale number for the real memory bars, **rmem** and **rmemc**, is the number of memory pages in the system. Each page is 4096 bytes in length. For the **sysact, gfx, intr, bdev, fault, pswap, net, netif, tlb** bars, the scale will show the calculated number of events per second which would fill the bar. The **cpu** and **wait** bars cannot be displayed as numeric bars. If the

bdev, **net** or **netif** bars are displayed as numeric bars then additional information is available.

The following options may be supported for each of the above bar types. If the option is unsupported, it will be silently ignored by *gr_osview*.

strip	- display as a strip chart
numeric	- display numeric values
max	- add a maximum value numeric meter
tracksum	- add an average value numeric meter
tracksum(n)	- add an average value meter with interval n
noborder	- suppress bar border
interval(n)	- set sample interval in base units
samples(n)	- set number of samples in strip chart
attack(f)	- set attack speed of bar
decay(f)	- set decay speed of bar
colors(c,...)	- set bar colors to use
maxcolor(b,f)	- set maximum value display colors
limcolor(b,f)	- set scale limit value display colors
sumcolor(b,f)	- set tracking sum display colors
lockscale(x)	- lock the scale value
ticks(i,n)	- set strip chart tick marks

A strip chart is a pictorial representation of a number of samples from the given bar, displayed vertically rather than horizontally. The header displays the total time covered by the bar, which is the interval times the number of samples. A numeric chart simply shows the monitored values as absolute numeric values rather then as pictorial values.

The **max** enables max value mode, while the **tracksum** option enables average value tracking mode. In the second form of **tracksum**, the argument specifies the number of basic intervals over which to average the value. This allows, for instance, a moving bar to react very quickly for good visual effect while the average value is computed over a longer interval for more accuracy. If the **noborder** option is given, then the bar border is suppressed.

The **interval** option specifies the update interval of the bar, in base interval units. The base interval is set as a global option; see below. The argument to this option indicates the number of base interval units in the interval for this bar. For instance, if the base interval is .2 seconds, then an argument of "5" would indicate a 1 second interval.

The **attack** option specifies the percentage upward change in the value of a bar parameter allowed per sample. This hysteresis is what makes each bar appear to move smoothly; changing the attack value can change the speed and appearance of a bar substantially. The value can range from 0.0 to 1.0; of course 0.0 indicates no movement of the bar ever, while 1.0 means instant change to the next value. Similarly, the **decay** option modifies the same behavior in the downward movement of the bar. The default value for both attack and decay is 0.2, or 20%.

The **colors** option modifies the colors used for the bar. Starting from the first parameter in a bar, the colors are modified until either all parameters are modified or no more colors are specified. Each color specified in the argument is a color map index to use for that color. The **maxcolor, limcolor** and **sumcolor** options set the background and foreground colors, respectively, of that type of display for this bar.

The **lockscale** option locks the scale value for the bar to the given value, disabling autoscaling. A locked scale is indicated by the scale value being inverted in the display. If the numeric argument to **lockscale** is suffixed by a 'k' or 'K', then the argument is multiplied by 1024, and a 'm' or 'M' suffix multiplies the argument by 1 meg (1024 times 1024).

The **ticks** option enables tick marks for the given strip. Tick marks are painted at the top of the bar, and follow the strip as it moves. The first form, *ticks*(*n*), paints a tick mark every *n* samples. The second form, *ticks*(*n*,*m*), paints a double-length tick mark every (*n***m*) samples, or every *m* normal tick marks.

Global options are introduced by a line with the special descriptor **opt** beginning the line. The following global options are accepted:

noborder	- suppress borders on all bars
arbsize	- allow arbitrary window sizing
width(w)	- set number of bars horizontally
interval(t)	- set base sample interval
colors(c,...)	- modify default color behavior
maxcolor(b,f)	- set colors for maximum value displays
limcolor(b,f)	- set colors for limit value displays
sumcolor(b,f)	- set colors for sum value displays
backcolor(c)	- set background color
frontcolor(c)	- set foreground color
font(f)	- set font to use
origin(x,y)	- set window origin
winsize(x,y)	- set window size

The **noborder** option suppresses borders on all bars, making each bar display seem to "hang in space". The **arbsize** option allows arbitrary sizing of the window; *gr_osview* may not always be able to properly scale text or draw to match the window size if it is too small. The acceptability of a too-small display is left to the user. The **width** option sets the number of bars horizontally to use. This value is one by default, meaning that a long, vertical display is used. The **interval** option sets the base sample interval. The argument is given in tenths of seconds. The default base interval is two tenths of a second.

The **colors** option sets the global color table, from which each bar selects its default colors, in the same manner as for an individual bar. The **limcolor**, **maxcolor** and **sumcolor** options set the background and foreground colors, respectively, of the limit value, maximum value and tracking sum value displays. The **backcolor** and **frontcolor** options set the general background or foreground colors respectively.

The **font** option sets the font to be used for text. This is a font name as known to the font manager. Default is "TimesBoldItalic". The **origin** option specifies the initial origin of the window in screen coordinates. The **winsize** option sets the initial window size in screen coordinates.

INTERPRETING THE DISPLAY

cpu The cpu bar statistically monitors the distribution of CPU cycles between user programs, the operating system, interrupt overhead, graphics and idle time. Computation-intensive loads will show large user times, while I/O or kernel service intensive loads will show up as increased system and interrupt time. If intensive graphics activity is under way, then the time spent waiting for the graphics hardware to context switch and the time spent waiting for the graphics FIFO to empty will consume a significant portion of the processor. *Gr_osview* perturbs this slightly, since it causes graphics context switches to occur.

The data is collected by sampling the program counter at the kernel clock frequency, which is 100HZ on the 4D series. This sampling is done automatically by the operating system; *gr_osview* simply collects the data and displays it.

wait The wait bar monitors the percentage of time that the system is idle due to waiting for outstanding I/O requests. If a processor can perform other useful work while waiting for an I/O request, then the processor is not viewed as idle (on the cpu bar) and the wait bar will not reflect any wait time. This bar is constructed by summing all the idle time for all processors in the system and displaying the percentage of that time that an I/O request was pending. Since in a multiprocessor system, there is no way to differentiate which processors have I/O pending, if there is any outstanding I/O request, and a processor goes idle, that processor (along with all other idle processors) will accumulate wait for I/O time rather than idle time.

The different forms of waiting are: **IO** refers to time spent waiting for traffic related to file system accesses (including local, remote, and mapped files, and normal file read and write). **Swap** refers to time spent waiting for paging and swapping operations to and from any swap devices. **Pio** refers to time spent waiting for physical IO to complete; for example, direct DMA to user space.

The information is collected in the same way as cpu time.

rmem This bar measures real memory usage. **Kernel** memory is memory allocated to the operating system and drivers. **Sysdata** describes filesystem meta-data usage, that is, information such as inodes, bitmaps, directories and the like that are used to find user data on disk. **Delwri** memory is occupied by disk pages which have not yet been written to disk, **userdata** is memory currently allocated to running processes, and **free** is memory not currently in use. Memory data is collected as absolute numbers rather than with statistical means, such as for cpu usage. Most bars listed here use this kind of accurate data, unless stated otherwise.

rmemc This bar is the same as the **rmem** bar, except that two fields are used to describe memory not currently in use: **freec** is unused memory which contains valid backing-store data and may be reclaimed by a process; **freeu** is unused memory which contains no usable data. Collecting the data for this bar consumes more cpu usage than **rmem**.

sysact The system activity bar measures a few of the important system activities, namely total system calls, process context switches, fork, exec and iget operations. **Fork** operations are initiated by a process when it wishes to create a new process. **Exec** operations are initiated by a process to overlay itself with a new process; this is how a new program image is loaded and run. Finally, an **iget** operation occurs whenever the state of a file is changed, for instance when it is opened. This is a crude measure of file system activity. It is important to note that *gr_osview* adds to the totals shown in this bar, since it to must perform at least some system calls, and causes some process context switches.

gfx This bar monitors graphics activity on the system. On most systems, the graphics hardware can interrupt the CPU when certain conditions occur; the first bar element measures the amount of such activity, named **intr**. Every time the processor switches to a new process which is using a window on the graphics screen, a graphics context switch occurs, labeled **swch**. The window manager and some graphics programs will interact with privileged code in the kernel to obtain some service; this is labeled **ioctl** after the system call used to perform the service. When a graphics program wishes to be synchronized with buffer swapping done during double buffering, it must wait for the next vertical retrace. The number of times this happens is labeled **swap**. Finally, the graphics pipeline is preceded by a FIFO buffer to smooth data movement into the pipeline. When the FIFO fills up, which it can with a fast processor, it interrupts the host, which waits until the FIFO has emptied somewhat before allowing the graphics process to continue drawing. The **fiwt** element describes the number of times this has happened. The **finowt** element describes the number of times the interrupt routine found that the FIFO was empty by the time it had saved state and entered the interrupt handler.

bdev This bar monitors the input/output activity to block devices. Block devices are usually those which hold filesystems, thus this bar measures filesystem throughput. The values are given in blocks per second, each block being 512 bytes in length. If displayed as a numeric bar, then the logical read and write rates for filesystem blocks which were already in memory are displayed, as well as the hit ratio, which is the ratio of logical operations to physical operations. Finally, the actual physical read/write rates are reported.

fault This bar monitors page fault activity. These are events that occur in managing the virtual address space of a process. The types of events are:

cpw These are *copy-on-write* events. This occurs when two or more processes are sharing a page, and one of them attempts to modify the page. To preserve the semantics of the UNIX fork() primitive, a copy of the shared page is made for the process, and it is then allowed to modify that page at will.

mod *Modified* faults occur the first time a process modifies a page, which tells the operating system that the page is dirty. For unmodified pages, the kernel can go to the filesystem to get a copy of the page, which saves the overhead of keeping a copy on swap during paging. A copy of all modified pages must be maintained in memory or on swap.

dmd *Demand-fill* faults occur the first time a process references a page with which there is no associated backing-store, for instance when first touching the BSS segment in a program. Since BSS is guaranteed to be zero, it is not kept in the object file, but is allocated to the process on demand by the kernel.

cache During paging, the kernel keeps a pool of pages which have been selected as candidates for stealing, and have backing store containing a copy of the page. This allows the kernel to respond to memory allocation needs quickly, while allowing a process to get back a page quickly if it touches it again. This pool is called the *page cache*. Each *cache* event indicates that a page fault occurred and the desired page was found in the page cache.

file *File* events occur when a page fault happens and a copy of the required page is fetched from the file system.

swap This event indicates that a page fault occurred, and the desired page was fetched from the disk swap area.

double
This event indicates that a second-level fault has occurred. On the 4D series, translation lookaside buffer (TLB) handling is performed entirely by software. This is done by looking up the missing page entry in a page table, and entering the virtual to physical mapping into the TLB. First-level faults are handled by extremely efficient low-level software. The page tables themselves are virtually mapped, so when the first level TLB handler attempts to load a page table entry, it may fault because the page table isn't mapped. This is a second-level fault, and must be repaired by high-level kernel routines.

pageref
This event indicates that a page fault occurred, but the page was actually still attached to the user address space. Reference faults are used by the operating system to keep accurate usage information when paging is imminent.

pswap The page swapping activity monitor bar measures access to the swap areas, for either swapin or swapout activity. During heavy paging, both swapin and swapout activity could be significant. On a system with only a single swap area on the same disk as the system and user data, the swapping rate will be effectively limited by the disk latency. Much higher swapping rates are possible with several swap areas configured across multiple disks and disk controllers. This may be necessary to achieve reasonable throughput in the face of many large jobs competing for main memory.

net The network activity bar measures some of the parameters of network performance. If displayed as a normal bar or strip chart, the **TCPinKb** portion measures the number of kilobytes per second received through the TCP/IP protocol, while **TCPoutKb** measures the number of kilobytes per second sent through the TCP/IP protocol.

If displayed as a numeric bar, two additional values are available: **IPpack** gives the number of IP packets sent or received through the Internet layer, while **IPforw** gives the number of those packets which were forwarded on to another machine. Internet packets are typically generated by NFS, which uses the UDP (User Datagram Protocol) for transferring data rather than TCP.

netif This bar monitors packets transmitted or received through the various network interfaces which may be present on a machine. If displayed as a regular bar, the number of packets transferred in or out over the interface is shown as **Ipack** or **Opack**. If displayed as a numeric bar, then three additional fields may be displayed: **Ierr** and **Oerr** measure the number of packets received in error or which had errors during transmission, while **coll** measures the number of packet collisions which occurred, which usually only happens on CSMA/CD interfaces.

Each machine supporting networking has a local loopback interface called *lo0*. This interface is not usually displayed unless specifically called out, and it does not have the **Ierr**, **Oerr** and **coll** fields.

tlb The **tlb** monitor bar measures translation lookaside buffer (TLB) activity on the system. **mpsync** counts how many times a request is made to flush all TLB entries on all processors and **vmwrap** indicates how many times **mpsync** is caused by a depletion of clean (with respect to the TLB) kernel virtual addresses. **flush** counts how many times an entire TLB is flushed on any one processor and **idwrap** shows how many times this happens because a processor's TLB ids have been depleted. **idget** monitors TLB id allocation and **idpurge** counts how many times a tlb id is forcefully removed from a process. Lastly, **vapurge** counts individual tlb entries being purged.

intr The interrupt monitor bar measures the interrupt rate in the system, and is broken out into VME interrupts and others, which are typically local interrupts to the CPU chip, such as serial I/O ports. The operating system clock also interrupts the CPU at 100HZ, thus at least this interrupt rate will always show as a background value. A large interrupt value, coupled with extremely sluggish performance, may indicate hardware problems, such as continuously interrupting device.

disk The disk monitor bar comes in two flavors, an EFS version and an "everything else" version. In the EFS version, the first two parameters specify the space taken by the used and free I-nodes on the volume. The third is the space used by files, and any remainder is free space. In the non-EFS version, only the space used by files is shown.

EXAMPLE

The following description file gives a layout identical to that used when the -a option of *gr_osview is given:*

```
cpu
rmem max tracksum
wait strip
sysact max tracksum
gfx max tracksum
```

FILES

/usr/sbin/gr_osview
$HOME/.grosview

BUGS

When using a strip chart display, and some other window obscures part of the strip chart, the bar will gradually turn to black. This is because an in-framebuffer copy operation is used to make the strip appear to move, and when part of the window is obscured there is nothing to copy. It is not clear that this bug will ever be fixed, because of the performance advantages of this style of update.

If the gr_osview window is redrawn, perhaps due to a repaint or resizing of the window, the next tick mark to be drawn on the strip may be drawn at an incorrect position. Following marks will have correct position and interval.

If the *arbsize* option is used, a tiny window can be drawn which is unintelligible. It is assumed that if the *arbsize* option is given, then a truly arbitrary size is desired.

A global colors option will only affect bars declared after the colors declaration.

NAME

gr_top – display processes having highest CPU usage in a window

SYNOPSIS

gr_top [**–i** interval] [**–n** lines] [**–B** color] [**–p** points]

DESCRIPTION

This command displays a sorted list of processes which are using some portion of the available CPU cycles on a machine. The display is updated periodically as specified by *interval.*

The following fields are displayed in order for each process: user name, process ID, process group ID, CPU usage as a percentage, processor currently executing the process, process priority, process size (in pages), resident set size (in pages), total amount of CPU time used by the process and the process name.

Two header lines are displayed. The first gives the machine name, the release and build date information for the operating system currently running, the processor type, the load averages (the average number of jobs in the run queue over the last 1, 5 and 15 minutes), the current time and the number of active processes. The next line consists of column titles describing the data displayed for each process.

OPTIONS

The following options are supported:

–i *interval*	This option sets the update interval in seconds; by default this is 5 seconds.
–n *lines*	This option sets the number of processes that can be displayed at once, and thus indirectly determines the window size.
–B *color*	This option sets the background color which will be used for the window. The *color* is an index into the system colormap.
–p *points*	This option sets the pointsize of the text used in the window, and indirectly controls the window size along with the number of lines specified. The default pointsize is 9 points.

NAME

grep – search a file for a pattern

SYNOPSIS

grep [options] limited regular expression [file ...]

DESCRIPTION

grep searches files for a pattern and prints all lines that contain that pattern. *grep* uses limited regular expressions (expressions that have string values that use a subset of the possible alphanumeric and special characters) like those used with *ed (1)* to match the patterns. It uses a compact non-deterministic algorithm.

Be careful using the characters $, *, [, ^, | , (,), and \ in the *limited regular expression* because they are also meaningful to the shell. It is safest to enclose the entire *limited regular expression* in single quotes *'...'*.

If no files are specified, *grep* assumes standard input. Normally, each line found is copied to standard output. The file name is printed before each line found if there is more than one input file.

Command line options are:

–b	Precede each line by the block number on which it was found. This can be useful in locating block numbers by context (blocks are 512 bytes long and number from 0).
–c	Print only a count of the lines that contain the pattern.
–i	Ignore upper/lower case distinction during comparisons.
–l	Print the names of files with matching lines once, separated by new-lines. Does not repeat the names of files when the pattern is found more than once.
–n	Precede each line by its line number in the file (first line is 1).
–s	Suppress error messages about nonexistent or unreadable files
–v	Print all lines except those that contain the pattern.

SEE ALSO

ed(1), egrep(1), fgrep(1), sed(1), sh(1).

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files (even if matches were found).

BUGS

Lines are limited to BUFSIZ characters; longer lines are truncated. BUFSIZ is defined in **/usr/include/stdio.h.**
If there is a line with embedded nulls, *grep* will only match up to the first null; if it matches, it will print the entire line.

NAME

groups – show group memberships

SYNOPSIS

groups [user]

DESCRIPTION

The *groups* command shows the groups to which you or the optionally specified user belong. Each user belongs to a group specified in the password file */etc/passwd* and possibly to other groups as specified in the file */etc/group*. Invoking *groups* without a *user* specified executes a *getgroups* system call, which returns the grouplist of which the process is currently a member. This list will only contain the effective group ID unless a *multgrps* call has been executed; in that case it will contain all groups in */etc/group* of which the user is a member. If you do not own a file but belong to the group which it is owned by then you are granted group access to the file.

When creating a new file, if the underlying filesystem was mounted with the BSD file creation semantics flag [see *fstab*(4)] or the **S_ISGID** bit is set [see *chmod*(2)] on the parent directory, then the group ID of the new file is set to the group ID of the parent directory, otherwise it is set to the effective group ID of the calling process.

The maximum number of groups of which a process may be a member is defined (as an lbootable option) in */usr/sysgen/master.d/kernel*, named NGROUPS_MAX.

SEE ALSO

multgrps(1), setgroups(2), open(2)

FILES

/etc/passwd, /etc/group

NAME

head – give first few lines

SYNOPSIS

head [–count] [file ...]

DESCRIPTION

This filter gives the first *count* lines of each of the specified files, or of the standard input. If *count* is omitted it defaults to 10.

EXAMPLE

```
head -6 filea fileb filec
```

will print out the first six lines of the three specified files. The filename will appear before each new set of head lines listed, if more than one file has been specified.

SEE ALSO

tail(1).

NAME

help – ask for help about SCCS error messages and commands

SYNOPSIS

help [args]

DESCRIPTION

Help finds information to explain a message from an SCCS command or explain the use of an SCCS command. Zero or more arguments may be supplied. If no arguments are given, *help* will prompt for one.

The arguments may be either message numbers (which normally appear in parentheses following messages) or command names, of one of the following types:

type 1	Begins with non-numerics, ends in numerics. The non-numeric prefix is usually an abbreviation for the program or set of routines which produced the message (e.g., **ge6**, for message 6 from the *get* command).
type 2	Does not contain numerics (as a command, such as **get**)
type 3	Is all numeric (e.g., **212**)

The response of the program will be the explanatory information related to the argument, if there is any.

When all else fails, try ‘‘help stuck’’.

EXAMPLE

```
help he1
```

prints the message for error number "he1".

FILES

/usr/lib/help	directory containing files of message text.
/usr/lib/help/helploc	file containing locations of help files not in **/usr/lib/help**.

DIAGNOSTICS

Use *help* (1) for explanations.

NOTES

help (1) is not a general help facility. It only knows about the SCCS commands and error messages generated by those commands.

SEE ALSO

admin(1), delta(1), edit(1), get(1)

NAME

hostid – set or print identifier of current host system

SYNOPSIS

hostid [*hexnum* | *hostname*]
hostid –h *hostname*

DESCRIPTION

The *hostid* command with no arguments prints the identifier of the current host in hexadecimal. This numeric value is expected to be unique across all hosts and is commonly set to the host's Internet address. The super-user can set the host ID by giving a hexadecimal argument or the hostname. If the argument is a hostname, *hostid* sets the ID to the name's Internet address listed in */etc/hosts*. *Hostid* sets the exit status to 0 if it successfully changed the host ID and to 1 if not. The script */etc/init.d/network* uses *hostid* to set the ID during system startup.

FILES

/etc/hosts	hostname-address database

SEE ALSO

gethostid(2), sethostid(2), network(1M), hosts(4)

NAME

hostname – set or print name of current host system

SYNOPSIS

hostname [–s] [nameofhost]

DESCRIPTION

Hostname prints the name of the current host. The *–s* option trims any domain information from the printed name. The super-user can set the hostname by supplying an argument; this is usually done in the network initialization script **/etc/init.d/network** using the contents of **/etc/sys_id**.

SEE ALSO

gethostname(2), sethostname(2)

I – J

NAME

ical – calendar

SYNOPSIS

ical

DESCRIPTION

ical is a calendar for use as a desk accessory. It displays one month at a time. Each click of the middle mouse button advances the month. Each click of the left mouse button makes the month go back by one.

NAME

ico – animate an icosahedron or other polyhedron

SYNOPSIS

ico [-display display] [-geometry geometry] [-r] [-d pattern] [-i] [-dbl] [-faces] [-noedges] [-sleep n] [-obj object] [-objhelp] [-colors color-list]

DESCRIPTION

Ico displays a wire-frame rotating polyhedron, with hidden lines removed, or a solid-fill polyhedron with hidden faces removed. There are a number of different polyhedra available; adding a new polyhedron to the program is quite simple.

OPTIONS

-r Display on the root window instead of creating a new window.

-d pattern
Specify a bit pattern for drawing dashed lines for wire frames.

-i Use inverted colors for wire frames.

-dbl Use double buffering on the display. This works for either wire frame or solid fill drawings. For solid fill drawings, using this switch results in substantially smoother movement. Note that this requires twice as many bit planes as without double buffering. Since some colors are typically allocated by other programs, most eight-bit-plane displays will probably be limited to eight colors when using double buffering.

-faces Draw filled faces instead of wire frames.

-noedges
Don't draw the wire frames. Typically used only when -faces is used.

-sleep *n* Sleep n seconds between each move of the object.

-obj *object*
Specify what object to draw. If no object is specified, an icosahedron is drawn.

-objhelp
Print out a list of the available objects, along with information about each object.

-colors *color color ...*
Specify what colors should be used to draw the filled faces of the object. If less colors than faces are given, the colors are reused.

ADDING POLYHEDRA

If you have the source to ico, it is very easy to add more polyhedra. Each polyhedron is defined in an include file by the name of objXXX.h, where XXX is something related to the name of the polyhedron. The format of the include file is defined in the file polyinfo.h. Look at the file objcube.h to see what the exact format of an objXXX.h file should be, then create your objXXX.h file in that format.

After making the new objXXX.h file (or copying in a new one from elsewhere), simply do a 'make depend'. This will recreate the file allobjs.h, which lists all of the objXXX.h files. Doing a 'make' after this will rebuild ico with the new object information.

SEE ALSO

X(1)

BUGS

A separate color cell is allocated for each name in the -colors list, even when the same name may be specified twice.

COPYRIGHT

NAME

id – print user and group IDs and names

SYNOPSIS

id

DESCRIPTION

id prints the user and group IDs and the corresponding names for the invoking process. If the effective and real IDs are different, both are printed.

If multiple groups are active (i.e. *multgrps*(1) has been called) all the groups of which the invoking user is a member are also listed.

FILES

/usr/bin/id	
/etc/passwd	system password file
/etc/group	system group file

SEE ALSO

logname(1), multgrps(1), newgrp(1) in the *User's Reference Manual*
setgroups(2), getgroups(2), getuid(2), initgroups(3X) in the *Programmer's Reference Manual*

NAME

ident – identify files

SYNOPSIS

ident file ...

DESCRIPTION

Ident searches the named files for all occurrences of the pattern *$keyword:...$*, where *keyword* is one of

Author
Date
Header
Locker
Log
Revision
Source
State

These patterns are normally inserted automatically by the RCS command *co*(1), but can also be inserted manually.

Ident works on text files as well as object files. For example, if the C program in file f.c contains

```
char rcsid[] = "$Header:  Header information $";
```

and f.c is compiled into f.o, then the command

```
ident  f.c  f.o
```

will print

```
f.c:
     $Header:  Header information $
f.o:
     $Header:  Header information $
```

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 3.1 ; Release Date: 83/04/04.
Copyright © 1982 by Walter F. Tichy.

SEE ALSO

ci(1), co(1), rcs(1), rcsdiff(1), rcsintro(1), rcsmerge(1), rlog(1), rcsfile(4).
Walter F. Tichy, ''Design, Implementation, and Evaluation of a Revision Control System,'' in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

NAME

imake – C preprocessor interface to the make utility

SYNOPSIS

imake [-D*define*] [-I*dir*] [-T*template*] [-f *filename*] [-s *filename*] [-e] [-v]

DESCRIPTION

Imake is used to generate *Makefiles* from a template, a set of *cpp* macro functions, and a per-directory input file called an *Imakefile*. This allows machine dependencies (such has compiler options, alternate command names, and special *make* rules) to be kept separate from the descriptions of the various items to be built.

OPTIONS

The following command line options may be passed to *imake*:

–D*define*
: This option is passed directly to *cpp*. It is typically used to set directory-specific variables. For example, the X Window System uses this flag to set *TOPDIR* to the name of the directory containing the top of the core distribution and *CURDIR* to the name of the current directory, relative to the top.

–I*directory*
: This option is passed directly to *cpp*. It is typically used to indicate the directory in which the *imake* template and configuration files may be found.

–T*template*
: This option specifies the name of the master template file (which is usually located in the directory specified with *–I*) used by *cpp*. The default is *Imake.tmpl*.

–f *filename*
: This option specifies the name of the per-directory input file. The default is *Imakefile*.

–s *filename*
: This option specifies the name of the *make* description file to be generated but *make* should not be invoked. If the *filename* is a dash (-), the output is written to *stdout*. The default is to generate, but not execute, a *Makefile*.

–e
: This option indicates the *imake* should execute the generated *Makefile*. The default is to leave this to the user.

–v
: This option indicates that *imake* should print the *cpp* command line that it is using to generate the *Makefile*.

HOW IT WORKS

Imake invokes *cpp* with any *–I* or *-D* flags passed on the command line and passes it the following 3 lines:

```
#define IMAKE_TEMPLATE "Imake.tmpl"
#define INCLUDE_IMAKEFILE "Imakefile"
#include IMAKE_TEMPLATE
```

where *Imake.tmpl* and *Imakefile* may be overridden by the *–T* and *–f* command options, respectively. If the *Imakefile* contains any lines beginning with a '#' character that is not followed by a *cpp* directive (**#include**, **#define**, **#undef**, **#ifdef**, **#else**, **#endif**, or **#if**), *imake* will make a temporary *makefile* in which the '#' lines are prepended with the string "/**/" (so that *cpp* will copy the line into the *Makefile* as a comment).

The *Imakefile* reads in file containing machine-dependent parameters (specified as *cpp* symbols), a site-specific parameters file, a file containing *cpp* macro functions for generating *make* rules, and finally the *Imakefile* (specified by INCLUDE_IMAKEFILE) in the current directory. The *Imakefile* uses the macro functions to indicate what targets should be built; *imake* takes care of generating the appropriate rules.

The rules file (usually named *Imake.rules* in the configuration directory) contains a variety of *cpp* macro functions that are configured according to the current platform. *Imake* replaces any occurrences of the string "@@" with a newline to allow macros that generate more than one line of *make* rules. For example, the macro

```
#define program_target(program, objlist)        @@\
program:        objlist                         @@\
        $(CC) -o $@ objlist $(LDFLAGS)
```

when called with *program_target(foo, foo1.o foo2.o)* will expand to

```
foo:    foo1.o foo2.o
        $(CC) -o $@ foo1.o foo2.o $(LDFLAGS)
```

On systems whose *cpp* reduces multiple tabs and spaces to a single space, *imake* attempts to put back any necessary tabs (*make* is very picky about the difference between tabs and spaces). For this reason, colons (:) in command lines must be preceded by a backslash (\).

USE WITH THE X WINDOW SYSTEM

The X Window System uses *imake* extensively, for both full builds within the source tree and external software. As mentioned above, two special variables, *TOPDIR* and *CURDIR* set to make referencing files using relative path names easier. For example, the following command is generated automatically to build the *Makefile* in the directory *lib/X/* (relative to the top of the sources):

```
% ../.././config/imake  -I../.././config \
       -DTOPDIR=../../. -DCURDIR=./lib/X
```

When building X programs outside the source tree, a special symbol *UseInstalled* is defined and *TOPDIR* and *CURDIR* are omitted. If the configuration files have been properly installed, the script *xmkmf(1)* may be used to specify the proper options:

```
% xmkmf
```

The command *make Makefiles* can then be used to generate *Makefiles* in any subdirectories.

FILES

/usr/tmp/tmp-imake.*nnnnnn*	temporary input file for cpp
/usr/tmp/tmp-make.*nnnnnn*	temporary input file for make
/lib/cpp	default C preprocessor

SEE ALSO

make(1)
S. I. Feldman *Make – A Program for Maintaining Computer Programs*

ENVIRONMENT VARIABLES

The following environment variables may be set, however their use is not recommended as they introduce dependencies that are not readily apparent when *imake* is run:

IMAKEINCLUDE
: If defined, this should be a valid include argument for the C preprocessor. E.g. "-I/usr/include/local". Actually, any valid *cpp* argument will work here.

IMAKECPP
: If defined, this should be a valid path to a preprocessor program. E.g. "/usr/local/cpp". By default, *imake* will use /lib/cpp.

IMAKEMAKE
: If defined, this should be a valid path to a make program. E.g. "/usr/local/make". By default, *imake* will use whatever *make* program is found using *execvp(3).*

BUGS

Comments should be preceded by "/**/#" to protect them from *cpp*.

AUTHOR

Todd Brunhoff, Tektronix and MIT Project Athena; Jim Fulton, MIT X Consortium

NAME

imged – small image editor

SYNOPSIS

imged *imgfile* [*xsize ysize* [**–rgb**]]

DESCRIPTION

imged is a pixel-based editor for making changes to image files stored in the SGI imglib format. Images may be stored in colormap, RGB, or greyscale format. is used.

imged may also be used to create a new colormap or RGB image by specifying an X and Y size on the command line. If the **–rgb** flag is given, the new image is created as RGB, otherwise it is created as a colormap image. New images are initialized with a solid grey background.

The *imged* window contains a scaled representation of the image being edited. The scale of the image can be changed by resizing the window. The pixels in the image will appear as squares, optionally separated by black lines. The title of the *imged* window indicates the type of image that is being edited.

The three mouse buttons all have a function within *imged* . The left and middle buttons both act as brushes. Each brush remembers a "current color". While the brush buttons are pressed, drawing will take place in the color of that brush. If the Shift key is held down, pushing a brush button causes the brush to change color to the color of the pixel directly under the cursor. The Alt key also modifies the operation of the brush buttons for RGB images. While the Alt key is held down, drawing will be performed in "blending mode". The pixels being drawn on will be blended with the color of the brush.

Pushing the right button causes a popup menu to appear, which can be used to change the mode of imged. Several options are available. The "no grid" option turns off the grid of black lines separating the pixels. Similarly, the "grid" option turns the grid on.

DRAWING MODES

imged has four different drawing modes: *sketch*, *rect*, *line*, and *smear*. To change the drawing mode, select the appropriate entry from the popup menu with the right mouse button.

sketch mode lets you change pixels directly under the cursor to the brush color when a brush button is depressed. Holding a brush button down while moving the mouse results in a sketch-like drawing.

rect mode	lets you create rectangles by pressing a brush button at one corner of the desired rectangle and holding the button while moving the cursor. When you release the button, the rectangle is drawn.
line mode	lets you draw straight lines by pressing a brush button at one end of the desired line segment and holding the button while moving the cursor. When you release the button, the line segment connecting the two points is drawn.
smear mode	is only available for use with RGB pictures. It lets you smear an area with the brush. The pixels surrounding the pixel the brush is on are used to determine the color used for that pixel.

HALFTONE

Often when editing small images, you need the ability to draw checker-board patterns of pixels. This can be done easily by selecting the "halftone" option from the popup menu. The right and middle mouse buttons will draw on pixels of opposite parity. To turn off the halftone mode, select the "solid" option from the menu.

WRITING CHANGES

To write out changes to an image, select the "write" option from the popup menu. The current state of the image will be written to the file be edited. Changes must be written to a file to be saved.

DIAGNOSTICS

Exit status is 1 if an error is encountered.

SEE ALSO

ipaste(1G)
Graphics Library Programming Guide

NAME

inform – display a message in a window

SYNOPSIS

inform [text]

DESCRIPTION

inform opens up a window on the graphics console containing the *text* and a Continue button. If the mouse is clicked in the Continue button, the window will go away.

BUGS

The window which appears is fixed size, and the text is not wrapped, so a lengthy message can be truncated.

SEE ALSO

confirm(1G)

NAME

install – install files in directories

SYNOPSIS

/etc/install *options file* ...

DESCRIPTION

Install copies regular *files* generated in a source tree into a target directory. It can also create directories, links and special files in a target directory. The target directory's pathname will be prefixed by the value of the **ROOT** environment variable. *Install* is most commonly invoked from makefiles (see *make*(1)).

If the **RAWIDB** environment variable is set, *install* creates no files in target directories; instead, it appends records describing the files that it would have created to the installation database (IDB) named by **$RAWIDB**. When generating an IDB, either *install* must be invoked under a directory named **src**, or the **SRC** environment variable must be set and must name an ancestor of the current working directory.

Install's options specify how to install, what type of file to create, and where to install in the target tree. Options are collected from the **INSTOPTS** environment variable, then from the command line. If incompatible options are specified in **INSTOPTS** and the command line, the command line options take precedence. Note that single-letter options cannot be concatenated after a hyphen. The options are:

–m *mode*	Set the mode of created files to *mode*, interpreted as an octal number. The default mode for regular files and directories is 755. The default mode for devices and named pipes is 666. This option cannot be used with **–ln** or **–lns**.
–u *owner*	Set the owner of created files to *owner*, which is interpreted first as a user name, then as a numeric user ID if it fails to match known user names. If the superuser invokes *install*, the default owner is **bin**. Otherwise the default owner is the effective user ID of the invoker. This option cannot be used with **–ln** or **–lns**.
–g *group*	Set the group of created files to *group*, which is interpreted first as a group name, then as a numeric group ID if it fails to match known group names. If the superuser invokes *install*, the default group is **bin**. Otherwise the default group is the effective group ID of the invoker. This option cannot be used with **–ln** or **–lns**.

–idb *attribute*	Add an IDB *attribute* to the records for files which *install* would have created in its normal mode. This option may occur several times among the option arguments.
–new	By default, *install* creates a regular file with the same modification time as its source. This option disables modification time preservation, creating a regular file with modification reflecting *install*'s final write to it.
–o	If a target exists, save it in the target directory with a hard link named OLD*file* .
–O	If the target exists, try to remove it. If it cannot be unlinked, save it in the same manner as **–o**.
–rawidb *idbpath*	Change *install*'s mode from file creation to IDB generation, so that it appends records to the file named by *idbpath*. This option overrides the **RAWIDB** environment variable.
–root *rootpath*	Set the string prefixed to all absolute pathnames created by install to *rootpath*. This option overrides the **ROOT** environment variable.
–s	Be silent. Older versions of *install* printed verbose information by default when installing. *Install* is now silent by default, but this option remains for compatibility.
–t	Symbolically link targets to sources when installing regular files.
–v	Be verbose. This option causes *install* to print a line telling source and target pathnames for each file installed.

Only one of the following options may be specified for a given invocation of *install*, to install non-regular files:

–blk *maj,min*	Create a block device node with major device number *maj* and minor number *min*. If *min* has the form *lowmin–highmin*, *install* creates block device nodes for minor numbers *lowmin* through *highmin*, inclusive.
–chr *maj,min*	Like **–blk**, but creates character device nodes.
–dir	Create directories named by concatenating **$ROOT** to the *file* arguments.

−fifo	Create named pipes named by the *file* arguments.
−ln *path*	Create hard links named by the *file* arguments to the node named by *path*.
−lns *path*	Create symbolic links named by the *file* arguments which point to *path*.

This option may be used only when installing regular files:

−src *path*	Use *path* as the source file's pathname when installing a regular file. This option is useful for renaming a source file in the target directory.

One of the following two options must be used unless installing directories with **−dir**:

−f *dir*	Install files in *dir*, which must be an writable directory.
−F *dir*	Like *−f*, but creates any directories in the *dir* pathname which do not exist.

If these options are used with **−dir**, *install* forms the pathnames of directories to create by concatenating $ROOT, *dir*, and *file*, separated by slashes.

EXAMPLES

To install several programs, issue:

```
install −f /etc mount umount
```

To install a file which has a different name in the target tree, and which might be executing during installation, use:

```
install −F /etc −src Install −O install
```

This invocation creates disk device nodes, along with necessary directories:

```
install −F /dev/dsk −m 600 −u root −g sys −blk 4,0−15 ips0d0s
```

SEE ALSO

make(1).

NAME

ipaste – display an image

SYNOPSIS

ipaste imgfile [-sx] [-n] [-o xorg yorg]

DESCRIPTION

ipaste is a window manager utility that displays images stored using the image library "libimg.a". *ipaste* reads pixel values, image dimensions and other relevant data from the file *imgfile* via calls to the image library. *ipaste* then displays the image in a window. On machines with few bit planes, ipaste dithers the image into the NeWS or Mex colormaps.

The **-sx** option pastes up the image with a white border or frame surrounding it.

The **-n** option pastes up the image without any border. When the image is viewed in this way, access to the window frame's stow and lightning bolt buttons is disabled as the frame and these buttons are not visible. However the NeWS window menu can still be popped up via the RIGHTMOUSE when the cursor is on top of the image, and the Stow and Quit menu items give the same results. Also moving the image around can still be accomplished either via the Move menu item, or by simply pressing down LEFTMOUSE or MIDDLEMOUSE and then moving the mouse.

The **-o xorg yorg** option automatically pastes the image at a position on the screen (absolute screen coordinates) that has as its origin (xorg, yorg).

NOTE: *ipaste* will NOT work with image files made using the old gl program *dither*. For these dithered types of images (8 bits deep) you will need to first use *fromdi* to convert them from dithered into RGB format, and then can use *ipaste* to display them. *fromdi.c* lives in */usr/people/4Dgifts/iristools/imgtools*, and the executable is in the moregltools subsystem on the EOE2 software distribution tape.

For more information about using the image library, see the *Graphics Library Programming Guide*.

NAME

ipcrm – remove a message queue, semaphore set or shared memory id

SYNOPSIS

ipcrm [*options*]

DESCRIPTION

ipcrm will remove one or more specified messages, semaphore or shared memory identifiers. The identifiers are specified by the following *options*:

–q *msqid*	removes the message queue identifier *msqid* from the system and destroys the message queue and data structure associated with it.
–m *shmid*	removes the shared memory identifier *shmid* from the system. The shared memory segment and data structure associated with it are destroyed after the last detach.
–s *semid*	removes the semaphore identifier *semid* from the system and destroys the set of semaphores and data structure associated with it.
–Q *msgkey*	removes the message queue identifier, created with key *msgkey*, from the system and destroys the message queue and data structure associated with it.
–M *shmkey*	removes the shared memory identifier, created with key *shmkey*, from the system. The shared memory segment and data structure associated with it are destroyed after the last detach.
–S *semkey*	removes the semaphore identifier, created with key *semkey*, from the system and destroys the set of semaphores and data structure associated with it.

The details of the removes are described in *msgctl*(2), *shmctl*(2), and *semctl*(2). The identifiers and keys may be found by using *ipcs*(1).

SEE ALSO

ipcs(1).
msgctl(2), msgget(2), msgop(2), semctl(2), semget(2), semop(2), shmctl(2), shmget(2), shmop(2) in the *Programmer's Reference Manual.*

NAME

ipcs – report inter-process communication facilities status

SYNOPSIS

ipcs [options]

DESCRIPTION

ipcs prints certain information about active inter-process communication facilities. Without *options*, information is printed in short format for message queues, shared memory, and semaphores that are currently active in the system. Otherwise, the information that is displayed is controlled by the following *options*:

–q Print information about active message queues.

–m Print information about active shared memory segments.

–s Print information about active semaphores.

If any of the options **–q**, **–m**, or **–s** are specified, information about only those indicated will be printed. If none of these three are specified, information about all three will be printed subject to these options:

–b Print biggest allowable size information. (Maximum number of bytes in messages on queue for message queues, size of segments for shared memory, and number of semaphores in each set for semaphores.) See below for meaning of columns in a listing.

–c Print creator's login name and group name. See below.

–o Print information on outstanding usage. (Number of messages on queue and total number of bytes in messages on queue for message queues and number of processes attached to shared memory segments.)

–p Print process number information. (Process ID of last process to send a message and process ID of last process to receive a message on message queues and process ID of creating process and process ID of last process to attach or detach on shared memory segments) See below.

–t Print time information. (Time of the last control operation that changed the access permissions for all facilities. Time of last *msgsnd* and last *msgrcv* on message queues, last *shmat* and last *shmdt* on shared memory, last *semop*(2) on semaphores.) See below.

−a Use all print *options*. (This is a shorthand notation for **−b**, **−c**, **−o**, **−p**, and **−t**.)

−C *corefile*
Use the file *corefile* in place of **/dev/kmem**.

−N *namelist*
The argument will be taken as the name of an alternate *namelist* (**/unix** is the default).

The column headings and the meaning of the columns in an *ipcs* listing are given below; the letters in parentheses indicate the *options* that cause the corresponding heading to appear; **all** means that the heading always appears. Note that these *options* only determine what information is provided for each facility; they do *not* determine which facilities will be listed.

T (all) Type of the facility:

q	message queue;
m	shared memory segment;
s	semaphore.

ID (all) The identifier for the facility entry.

KEY (all) The key used as an argument to *msgget*, *semget*, or *shmget* to create the facility entry. (Note: The key of a shared memory segment is changed to **IPC_PRIVATE** when the segment has been removed until all processes attached to the segment detach it.)

MODE (all) The facility access modes and flags: The mode consists of 11 characters that are interpreted as follows:
The first two characters are:

R	if a process is waiting on a *msgrcv*;
S	if a process is waiting on a *msgsnd*;
D	if the associated shared memory segment has been removed. It will disappear when the last process attached to the segment detaches it;
C	if the associated shared memory segment is to be cleared when the first attach is executed;
−	if the corresponding special flag is not set.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to the owner's permissions; the next to permissions of others in the user-group of the facility entry; and the last to all others. Within each set, the first character indicates permission to read, the second character indicates permission to write or alter the facility entry, and the last character is currently unused.

The permissions are indicated as follows:

r	if read permission is granted;
w	if write permission is granted;
a	if alter permission is granted;
–	if the indicated permission is *not* granted.

OWNER	(all)	The login name of the owner of the facility entry.
GROUP	(all)	The group name of the group of the owner of the facility entry.
CREATOR	(a,c)	The login name of the creator of the facility entry.
CGROUP	(a,c)	The group name of the group of the creator of the facility entry.
CBYTES	(a,o)	The number of bytes in messages currently outstanding on the associated message queue.
QNUM	(a,o)	The number of messages currently outstanding on the associated message queue.
QBYTES	(a,b)	The maximum number of bytes allowed in messages outstanding on the associated message queue.
LSPID	(a,p)	The process ID of the last process to send a message to the associated queue.
LRPID	(a,p)	The process ID of the last process to receive a message from the associated queue.
STIME	(a,t)	The time the last message was sent to the associated queue.
RTIME	(a,t)	The time the last message was received from the associated queue.
CTIME	(a,t)	The time when the associated entry was created or changed.
NATTCH	(a,o)	The number of processes attached to the associated shared memory segment.
SEGSZ	(a,b)	The size of the associated shared memory segment.
CPID	(a,p)	The process ID of the creator of the shared memory entry.
LPID	(a,p)	The process ID of the last process to attach or detach the shared memory segment.
ATIME	(a,t)	The time the last attach was completed to the associated shared memory segment.
DTIME	(a,t)	The time the last detach was completed on the associated shared memory segment.
NSEMS	(a,b)	The number of semaphores in the set associated with the semaphore entry.

OTIME	(a,t)	The time the last semaphore operation was completed on the set associated with the semaphore entry.

FILES

/unix	system namelist
/dev/kmem	memory
/etc/passwd	user names
/etc/group	group names

SEE ALSO

msgop(2), semop(2), shmop(2) in the *Programmer's Reference Manual.*

BUGS

Things can change while *ipcs* is running; the picture it gives is only a close approximation to reality.

NAME

isSuper – supertype checking utility for use with file type rules

SYNOPSIS

isSuper supertype testtype [file.ctr]

DESCRIPTION

isSuper is used to check if *testtype* has a supertype of type *supertype* defined in the *.ctr* file *file.ctr* or a default of */usr/lib/filetype/workspace.ctr*.

SEE ALSO

Programming the IRIS WorkSpace

NAME

join – relational database operator

SYNOPSIS

join [options] file1 file2

DESCRIPTION

join forms, on the standard output, a join of the two relations specified by the lines of *file1* and *file2*. If *file1* is –, the standard input is used.

File1 and *file2* must be sorted in increasing ASCII collating sequence on the fields on which they are to be joined, normally the first in each line [see *sort*(1)].

There is one line in the output for each pair of lines in *file1* and *file2* that have identical join fields. The output line normally consists of the common field, then the rest of the line from *file1*, then the rest of the line from *file2*.

The default input field separators are blank, tab, or new-line. In this case, multiple separators count as one field separator, and leading separators are ignored. The default output field separator is a blank.

Some of the below options use the argument *n*. This argument should be a 1 or a 2 referring to either *file1* or *file2*, respectively. The following options are recognized:

–a*n* In addition to the normal output, produce a line for each unpairable line in file *n*, where *n* is 1 or 2.

–e *s* Replace empty output fields by string *s*.

–j*n m* Join on the *m*th field of file *n*. If *n* is missing, use the *m*th field in each file. Fields are numbered starting with 1.

–o *list* Each output line comprises the fields specified in *list*, each element of which has the form *n.m*, where *n* is a file number and *m* is a field number. The common field is not printed unless specifically requested.

–t*c* Use character *c* as a separator (tab character). Every appearance of *c* in a line is significant. The character *c* is used as the field separator for both input and output.

EXAMPLE

The following command line will join the password file and the group file, matching on the numeric group ID, and outputting the login name, the group name and the login directory. It is assumed that the files have been sorted in ASCII collating sequence on the group ID fields.

```
join –j1 4 –j2 3 –o 1.1 2.1 1.6 –t: /etc/passwd /etc/group
```

SEE ALSO

awk(1), comm(1), sort(1), uniq(1).

BUGS

With default field separation, the collating sequence is that of **sort –b**; with **–t**, the sequence is that of a plain sort.

The conventions of *join*, *sort*, *comm*, *uniq* and *awk*(1) are wildly incongruous.

Filenames that are numeric may cause conflict when the **-o** option is used right before listing filenames.

NAME

jot – a simple mouse-based text editor

SYNOPSIS

jot [**–f** fontname] [files...]

DESCRIPTION

jot is a simple editor that uses the mouse to cut, copy, and paste text, and to position the cursor. *jot* also allows you to perform simple searches.

If you specify the **–f** option followed by a font specification, jot will use the specified font to display the text. For example:

```
        jot -f Courier14
```

opens a *jot* window with a 14-point Courier font.

If you specify a set of text *files* separated by spaces on the command line, jot will open that file for editing; otherwise it will open a new file.

The *jot* menu provides the following facilities:

Cut Remove selected text to the cut buffer. This text can be sent to a *wsh* window or to another *jot* window, as well as the window it was cut from. This function can also be accessed using the F2 function key on the keyboard.

Copy Copy selected text to the cut buffer. This text can be sent to a *wsh* window or to another *jot* window, as well as the window it was copied from. This function can also be accessed using the F3 function key on the keyboard.

Paste Transfer text from the cut buffer to the current text cursor location (the *current point*). If text is selected when the paste is made, the selected text is replaced with the text from the buffer. This text could have been cut/copied from any *jot* or *wsh* window. This function can also be accessed using the F4 function key on the keyboard.

Search Search the document for the first instance of a string. Choosing 'Search' generates a notifier window that requests a string. *jot* searches sequentially from the current point through the rest of the file for that string, selecting the first instance it comes across. To find the next instance, you must choose 'Select' again.

Open Open a text file. Choosing 'Open' generates a notifier window that requests a file name. *jot* opens a new window with the new file in it, keeping the old one open as well.

SELECTING TEXT

Select text with the left mouse button by performing the following actions:

click	Sets the new current point to wherever the mouse (arrow) cursor is pointing in the text.
click-hold-release	This action selects all text between the the point where the left mouse button is first clicked down and the point where it is released.
click-click	Double-clicking the left mouse causes the word over which the mouse cursor is clicked to be selected.
shift&click-hold-release	If you have already selected a block of text, you can extend your selection either forwards or backwards by holding down the shift key while clicking the left mouse.
alt&click-hold-release	Same as *click-hold-release*, except that an implicit copy is performed when the left mouse button is released. You can use *alt&shift&click* as well.

SCROLL BARS

jot uses horizontal and vertical scroll bars with proportional thumbs.

Clicking on the arrows with the left mouse button causes the *jot* window to scroll one character up, down, left, or right. Holding down the left button over any arrow causes the scrolling to auto-repeat. Holding down the shift key while clicking on any arrow causes the window to scroll one full page of text.

The scroll bars' thumbs can also be used to scroll through the *jot* window by clicking and holding down the left mouse button while dragging the thumb with the mouse cursor. The size of the thumbs are variable, and indicate the percentage of total text that is visible in the window.

BUGS

The horizontal scrollbar can only handle lines a maximum of 200 characters across.

SEE ALSO

jotview(1G), wsh(1G)

NAME

jotview – a simple mouse-based text viewer

SYNOPSIS

jotview [**–f** fontname] [files...]

DESCRIPTION

jotview is a simple text viewer that uses the mouse to copy text and position the cursor. *jotview* also allows you to perform simple searches.

If you specify the **–f** option followed by a font specification, jot will use the specified font to display the text. For example:

```
jot -f Courier14 foo
```

opens a *jotview* window containing the file *foo* using a 14-point Courier font.

The *jotview* menu provides the following facilities:

Copy Copy selected text to the cut buffer. This text can be sent to a *wsh* window or to a *jot* window. This function can also be accessed using the F3 function key on the keyboard.

Search Search the document for the first instance of a string. Choosing 'Search' generates a notifier window that requests a string. *jotview* searches sequentially through the file for that string, selecting the first instance it comes across. To find the next instance, you must choose 'Select' again.

Open Open a text file. Choosing 'Open' generates a notifier window that requests a file name. *jotview* opens a new window with the new file in it, keeping the old one open as well.

SELECTING TEXT

Select text with the left mouse button by performing the following actions:

click	Sets the location of the text cursor (the *current point*) to wherever the mouse (arrow) cursor points in the text.
click-hold-release	This action selects all text between the the point where the left mouse button is first clicked down and the point where it is released.
click-click	Double-clicking the left mouse causes the word over which the mouse cursor is clicked to be selected.

shift&click-hold-release	If you have already selected a block of text, you can extend your selection either forwards or backwards by holding down the shift key while clicking the left mouse.
alt&click-hold-release	Same as *click-hold-release*, except that an implicit copy is performed when the left mouse button is released. You can use *alt&shift&click* as well.

SCROLL BARS

jotview uses horizontal and vertical scroll bars with proportional thumbs.

Clicking on the arrows with the left mouse button causes the *jotview* window to scroll one character up, down, left, or right. Holding down the left button over any arrow causes the scrolling to auto-repeat. Holding down the shift key while clicking on any arrow causes the window to scroll one full page of text.

The scroll bars' thumbs can also be used to scroll through the *jotview* window by clicking and holding down the left mouse button while dragging the thumb with the mouse cursor. The size of the thumbs are variable, and indicate the percentage of total text that is visible in the window.

BUGS

The horizontal scrollbar can only handle lines a maximum of 200 characters across.

SEE ALSO

jot(1G), wsh(1G)

NAME

journalchest – 4Sight event record and playback toolchest

SYNOPSIS

journalchest

DESCRIPTION

journalchest creates a toolchest allowing access to the 4Sight journalling package. Journalling allows you to capture 4Sight mouse and keyboard events onto a file, and then play the file back. This results in 4Sight acting like a player piano, faithfully duplicating the original user actions in real time.

Journalling permits continuous replaying of a given file. Playback can be interrupted at any time by clicking one of the mouse buttons.

The journalling package also includes playback speed control, which allows you to slow down or speed up the playback rate.

USAGE

There are five menu items in the Journalling chest:

Control Panel - Brings up the journalling control panel

Start Recording - Starts recording on the current Record file

Stop Recording - Signals the end of recording

Playback - Starts playback of the current Playback file

Remove Journalling - Removes the Journalling chest and the journalling package from 4Sight.

Playback can be interrupted at any time by hitting one of the mouse keys.

Selecting the Control Panel item brings up a control panel window. It contains the following items:

RECORD, STOP, and PLAY buttons: These buttons perform the same function as the corresponding menu items. They also light up to indicate what action is currently taking place. They can be used interchangeably with the menu items.

Record File: This text item allows you to specify the current file to record onto. It can be any valid filename on the server machine.

Relative pathnames are taken to be relative to the directory that 4Sight was started from. The default for the Record file is */tmp/NeWS.journal.*

Playback File: The current file to playback from. It has the same characteristics as the Record File.

Play Forever toggle switch: If this switch is on then journalling will automatically repeat playing the Playback File.

Playback Speed: Slider that scales the playback time. Positive values make playback speed up, negative values make playback slow down. This facility is dependent on the speed of the underlying hardware. It is not calibrated between different machines.

Done: The Done button will hide the Control Panel. It can be brought back up by selecting the *Control Panel* menu item. Selecting 'Quit' from the panel's Window menu has the same effect.

TIPS FOR USING JOURNALLING

When creating journals that will be replayed repeatedly, it is important to get rid of whatever windows you have created at the end of the journal. The state of the screen should be just as it was when the journal was begun. Otherwise, the 4Sight server will eventually run out of memory because you are continually creating new windows.

There is a noticeable variation in performance on 4Sight (NeWS) running on different kinds of machines; this means that playing back a script recorded on a fast machine might not always work correctly on a slower machine. The Playback Speed Control will allow you to adapt playback speed of a given script to a fairly wide range of machines; unfortunately, this requires a bit of trial and error.

Care must be taken when recording sequences that contain invocations of Unix programs, particularly when starting new applications. The mouse must not be clicked until the bounding box is up on the screen. If the mouse is clicked early, the wrong window sizing will be made on playback, leading to unpredictable behavior due to the window not being where it was when recording.

FILES

${NEWSHOME}/lib/NeWS/journal.ps	Contains the low level journalling code, the control panel and menu code, and state button Lite Item code used by the control panel.
/usr/sbin/journalchest	Contains the postscript code for generating the Journalling chest.

SEE ALSO

journalplay(1)

BUGS

The unpredictable behavior of playback due to the non-deterministic Unix scheduling mechanism and general operating environment make reliance on the journalling package for critical functions inadvisable.

If a GL client issues a setvaluator(3g) call on MOUSEX or MOUSEY while journalling is playing back a mouse button down the journalling package thinks the user wants to stop playback. This happens because the NeWS server has no way to distinguish between an event generated by setvaluator and one generated by moving the mouse.

NAME

journalplay, journalrecord, journalend – 4Sight event recording and playback from an IRIX shell

SYNOPSIS

journalplay [**–w**] [**–f**] [**–p** speed] [journalfile]
journalrecord [filename]
journalend

DESCRIPTION

journalplay starts playback of a journalling script from the specified journalling file. If no file name is provided on the command line, the default journalling file, */tmp/NeWS.journal* is used.

The following options are available on the command line:

–w Waits until the playback is finished before returning control to the shell; this is useful when *journalplay* is called from within a shell script.

–f Plays the script forever in a continuous loop.

–p *speed* Sets the speed multiplier; the default value for *speed* is 1.

journalrecord records a journalling script in the specified file. If no file is specified, it will record the script in */tmp/NeWS.journal.*

journalend terminates any currently recording script.

SEE ALSO

journalchest(1)

BUGS

The unpredictable behavior of playback due to the non-deterministic Unix scheduling mechanism and general operating environment make reliance on the journalling package for critical functions inadvisable.

K – L

NAME

kill – terminate a process

SYNOPSIS

kill [–sig] PID ...
kill –l
/bin/kill –L

DESCRIPTION

kill sends signal 15 (terminate) to the specified processes. This will normally kill processes that do not catch or ignore the signal. The process number of each asynchronous process started with & is reported by the shell (unless more than one process is started in a pipeline, in which case the number of the last process in the pipeline is reported). Process numbers can also be found by using *ps*(1).

The details of the kill are described in *kill*(2). For example, if process number 0 is specified, all processes in the process group are signaled.

The killed process must belong to the current user unless he is the super-user.

If a signal number preceded by – is given as first argument, that signal is sent instead of terminate (see *signal*(2)). In particular ''kill –9 ...'' is a sure kill. The signal names are listed by ''kill –l'', and are as given in */usr/include/sys/signal.h,* stripped of the common SIG prefix. The command ''/bin/kill –L'' gives a more verbose listing of the signal names and numbers.

SEE ALSO

csh(1), ps(1), sh(1).
kill(2), signal(2) in the *Programmer's Reference Manual.*

NAME

last – indicate last logins of users and terminals

SYNOPSIS

last [**–N**] [**–f** *file*] [name ...] [tty ...]

DESCRIPTION

Last will look in the *wtmp* file which records all logins and logouts for information about a user, a terminal or any group of users and terminals. Arguments specify names of users or terminals of interest. Names of terminals may be given fully or abbreviated. For example 'last 0' is the same as 'last tty0'. If multiple arguments are given, the information which applies to any of the arguments is printed. For example 'last root console' would list all of "root's" sessions as well as all sessions on the console terminal. *Last* will print the sessions of the specified users and terminals, most recent first, indicating the times at which the session began, the duration of the session, and the terminal which the session took place on. *Last* indicates if the session is still continuing or was cut short by a reboot,

Last with no arguments prints a record of all logins and logouts, in reverse order. The –N option limits the report to N lines, where N is a number.

Last ordinarily does not display uninteresting records associated with starting and stopping a login. The **–a** option displays those records.

The **–f** option can be used to examine another file, such as */etc/OLDwtmp*, in which old records are kept.

last is interrupted, it indicates how far the search has progressed in *wtmp*. If interrupted with a quit signal (generated by a control-\) *last* indicates how far the search has progressed so far, and the search continues.

wtmp must exist for the *last* command to be useful. If *wtmp* does not exist, the system will not create it. It must be created by the system administrator.

FILES

/etc/wtmp	login data base
/etc/xwtmp	extended login data base

SEE ALSO

wtmp(4)

NAME

launch – graphical utility to enter arguments and invoke commands

SYNOPSIS

launch [–**h** header] [–**m** message] [–**t** trailer] [–**c** command]

DESCRIPTION

launch is used to invoke commands through a window that contains a text edit field to allow for command completion. The options to launch are as follows:

–**h** header	puts the header *header* on the command that is used to invoke the command but is not displayed.
–**t** trailer	puts the trailer *trailer* on the command that is used to invoke the command but is not displayed.
–**m** message	displays the message specified by *message* in the window.
–**c** command	sets the command invoked to *command.*

When no arguments are specified, *launch* comes up with an empty text field waiting for input.

If the launched program requires a tty, you must call *launch* with the –**h** option, followed by an appropriate invocation of *winterm.* The following is an example of launch as used with *mail.*

```
launch -h winterm -c mail
```

SEE ALSO

winterm(1)

Programming the IRIS WorkSpace

NAME

ld, uld – MIPS link editor and ucode link editor

SYNOPSIS

ld [option] ... file ...
uld [option] ... file ...

DESCRIPTION

Ld, the MIPS link editor, links MIPS extended *coff* object files. The archive format understood by *ld* is the one created by the MIPS archiver *ar*(1).

The *ld* command combines several object files into one, performs relocation, resolves external symbols, and supports symbol table information for symbolic debugging. In the simplest case, the names of several object *files* are given. *Ld* combines them, producing an object module that can be executed or used as input for a subsequent *ld* run. (In the latter case, the **–r** option must be given to preserve the relocation entries.) The output of *ld* is left in **a.out**. By default, this file is executable if no errors occurred during the load.

The argument object files are concatenated in the order specified. The entry point of the output is the beginning of the text segment (unless the **–e** option is specified).

The *uld* command combines several ucode object files and libraries into one ucode object file. It ''hides'' external symbols for better optimizations by subsequent compiler passes. The symbol tables of *coff* object files loaded with ucode object files are used to determine what external symbols not to ''hide'' along with files specified by the user that contain lists of symbol names.

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. The library (archive) symbol table (see *ar*(1)) is a hash table and is searched to resolved external references that can be satisfied by library members. The ordering of library members is unimportant.

The following options are recognized by both *ld* and *uld* . Options used by *ld* but not by *uld* are ignored by *uld.* Options used by *uld* but not by *ld* are ignored by *ld.* Any option can be preceded by a '**k**' (for example **–ko** *outfile*) and except for **–kl***x* have the same meaning with or without the preceding '**k**'. This is done so that these options can be passed to both link editors through compiler drivers.

When searching for libraries the default directories to search in are /lib, /usr/lib and /usr/local/lib.

The symbols 'etext', 'edata', 'end', '_ftext', '_fdata', '_fbss', '_gp', '_procedure_table', '_procedure_table_size', and '_procedure_string_table' are reserved. These loader defined symbols have their values set as described in *end*(3). It is erroneous to define these symbols.

−o *outfile*
: Produce an output object file by the name *outfile*. The name of the default object file is **a.out**. In the event that *outfile* has undefined references at the end of the link editor run, and the **−r** option was not used, and the **−aoutkeep** option was used *outfile* will be renamed **a.out**. If there are undefined references and the **−aoutkeep** option was not used there will be no output from *ld*. **a.out** will, if created, contain relocation information, as if the **−r** option was used.

−l*x*
: Search a library **lib***x***.a**, where *x* is a string. A library is searched when its name is encountered, so the placement of a **−l** is significant.

−kl*x*
: Search a library **lib***x***.b**, where *x* is a string. These libraries are intended to be ucode object libraries. In all other ways, this option is like the **−l***x* option. Please note that this switch is non-standard, and may not be supported across product lines.

−L *dir*
: Change the algorithm of searching for **lib***x***.a** or **lib***x***.b** to look in *dir* before looking in the default directories. This option is effective only if it precedes the **−l** options on the command line. Repeating for emphasis: *dir* must be a directory.

−L*dir*
: The same as **−L** *dir* but the **−L***dir* form is inconsistent with *intro*(1).

−L
: Change the algorithm of searching for **lib***x***.a** or **lib***x***.b** to **never** look in the default directories. This is useful when the default directories for libraries should not be searched and only the directories specified by **−L***dir* are to be searched. This option may be dropped from some future release in favor of the **−nostdlib** option.

−nostdlib
: Change the algorithm of searching for **lib***x***.a** or **lib***x***.b** to **never** look in the default directories. This is useful when the default directories for libraries should not be searched and only the directories specified by **−L dir** are to be searched.

−B*string*
: Append *string* to the library names created for the **−l***x* and **−kl***x*.

−p *file* Preserve (don't ''hide'') the symbol names listed in *file* when loading ucode object files. The symbol names in the file are separated by blanks, tabs, or newlines. Please note that this switch is non-standard, and may not be supported across product lines.

−aoutkeep
: In case there undefined references the output will be in the file **a.out** with relocation information, as if **−r** had been given.

−s Strip the symbolic information from the output object file.

−x Do not preserve local (non-*.globl*) symbols in the output symbol table; enter external and static symbols only. This option saves some space in the output file.

−r Retain relocation entries in the output file. Relocation entries must be saved if the output file is to become an input file in a subsequent *ld* run. This option also prevents final definitions from being given to common symbols, and suppresses the 'undefined symbol' diagnostics.

−d Force definition of common storage and define loader defined symbols even if -r is present.

−u *symname*
: Enter *symname* as an undefined in the symbol table. This is useful for loading entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

−U Undefined symbols are not considered an error when this is supplied. As a result, one can create an executable with undefined symbols. Referencing an undefined symbol during execution may cause a segmentation fault and a coredump.

−z Arrange that the process pages are loaded on demand from the resulting executable file (0413, or ZMAGIC, format) rather than preloaded, and the text pages shared among all users. This is the default.

−n Arrange that when the output file is executed, the entire text and data portions of the executables are preloaded. This is an NMAGIC (0410) format. This involves moving the data areas up to the first possible *pagesize* byte boundary following the end of the text.

−N Place the data section immediately after the text and do not make the text portion read only or sharable. This is an OMAGIC (0407) file. See **NOTES** section for limitations on OMAGIC usage.

−T *num*
Set the text segment origin. The argument *num* is a hexadecimal number. It should be rounded to a multiple of hexadecimal 1000 for use with -A.

−D *num*
Set the data segment origin. The argument *num* is a hexadecimal number. It should be rounded to a multiple of hexadecimal 1000 for use with -A. Please note that this switch is non-standard, and may not be supported across product lines.

−B *num*
Set the bss segment origin. The argument *num* is a hexadecimal number. Please note that this switch is non-standard, and may not be supported across product lines.

−e *epsym*
Set the default entry point address for the output file to be that of the symbol *epsym.*

−m Produce a map or listing of the input/output sections on the standard output (UNIX System V-like map).

−M Produce a primitive load map, listing the names of the files that will be loaded (UNIX 4.3bsd-like map).

−v Set verbose mode. Print the name of each file as it is processed.

−y*sym* Indicate each file in which *sym* appears, *sym*'s type and whether the file defines or references *sym*. Many such options may be given to trace many symbols.

−V Print a message giving information about the version of *ld* being used.

−VS *num*
Use *num* as the decimal version stamp to identify the a.out file that is produced. The version stamp is stored in the optional and symbolic headers.

−f *fill* Set the fill pattern for ''holes'' within an output section. The argument *fill* is a four-byte hexadecimal constant.

−G *num*
The argument *num* is taken to be a decimal number that is the largest size in bytes of a *.comm* item that is to be allocated in the small bss section for reference off the global pointer. The default is 8

bytes. Please note that this switch is non-standard, and may not be supported across product lines.

–bestGnum
Calculate the best –G *num* to use when compiling and linking the files which produced the objects being linked. Using too large a number with the –G *num* option may cause the gp (global-pointer) data area to overflow; using too small a number may reduce your program's execution speed. Please note that this switch is non-standard, and may not be supported across product lines.

–count, –nocount, –countall
These options control which objects are counted as recompilable for the best –G *num* calculation. By default, the **–bestGnum** option assumes you can recompile everything with a different –G *num* option. If you cannot recompile certain object files or libraries (because, for example, you have no sources for them), use these options to tell the link editor to take this into account in calculating the best –G *num* value. **–nocount** says that object files appearing after it on the command line cannot be recompiled; **–count** says that object files appearing after it on the command line can be recompiled; you can alternate the use of **–nocount** and **–count. –countall** overrides any **–nocount** options appearing after it on the command line. Please note that this switch is non-standard, and may not be supported across product lines.

–b Do not merge the symbolic information entries for the same file into one entry for that file. This is only needed when the symbolic information from the same file appears differently in any of the objects to be linked. This can occur when object files are compiled, by means of conditional compilation, with an apparently different version of an include file.

–jmpopt and **–nojmpopt**
Fill (**–jmpopt**) or do not fill (**–nojmpopt**) the delay slots of jump instructions with the target of the jump and adjust the jump offset to jump past that instruction. This is always disabled for debugging (when **–g**, **–g1**, or **–g2** is present). When this option is enabled, it requires that all of the loaded program's text be in memory and could cause the loader to run out of memory. The default is **–nojmpopt**.

–g or **–g[0123]**
These debugging flags are accepted but their only effect is that **–g**, **–g1**, and **–g2** disable **–jmpopt**. **–g0** and **–g3** have no effect.

–A *file* This option specifies incremental loading – linking is done in a manner such that the resulting object may be read into an already executing program. The argument *file* is the name of a file whose symbol table will be taken as a basis on which to define additional symbols. Only newly linked material will be entered into the text and data portions of the a.out. The symbol table, however, will reflect every symbol defined both before and after the incremental load.

This argument must appear before any other object file in the argument list. The –T and –D options may be used with –A to indicate the address at which text and data sections of the newly linked segment will commence. This address must be correct multiple for the resulting object type. (See the **NOTES** section.) The default resulting object type is NMAGIC and the default starting address of the text is the old value of **end** rounded to 4096 (a page, 1000 in hexadecimal). The default starting address of the data is 10000000 (hexadecimal). When this file's sections are is read into an already executing file, they must be read into the addresses they were linked into. (That is, they must be read into the address given by s_vaddr in each section's header. See *<scnhdr.h>* for section header information.) See *ldfcn*(4) for information on reading the file. Remember that bss and sbss sections are not to be read in, but are simply to be initialized to zero. All other objects except the *file* argument to the –A option must be compiled –G 0 and this sets –G 0 for linking.

The following options are used by the command *mkshlib*(1) and are not intended for general use.

–c Create a target shared library object file. This is a LIBMAGIC file (443 format). The objects linked must be compiled with –G 0 and this sets –G 0 for linking. This file is demand paged. The organization of the file is unusual in two ways: first, the first page of the text segment is used by the header; actual text begins on the second page. Second, the data sections are organized differently from the other type of objects.

–i *file* The *.text* section of *file* is moved into the *.init* section of the resulting object file.

FILES

/lib/lib*.a	
/usr/lib/lib*.a	
/usr/local/lib/lib*.a	libraries
a.out	output file

SEE ALSO

cc(1), pc(1), f77(1), as(1), ar(1), end(3)

NOTES

Only ZMAGIC and NMAGIC objects can be run on the system; OMAGIC objects can only be run as stand-alone objects.

An object's segments must not overlap and all of the object's addresses must be less than 0x80000000. The stack starts at 0x80000000 and grows down through lower addresses therefore space should be left for it. For ZMAGIC and NMAGIC objects, the default text segment address is 0x00400000 and the default data segment address is 0x10000000. For OMAGIC objects, the default text segment address is 0x10000000 with the data segment immediately following the text segment. The default for all types of objects is that the bss segment follows the data segment. The segments must currently be on 2 megabyte boundaries though it is recommended that they be on 4 megabyte boundaries.

When creating an executable object, a.out, */usr/lib/crt1.o* Should be the first object loaded and */usr/lib/crtn.o* should be last following any libraries. The compiler drivers (*cc*(1), *f77*(1), and *pc*(1)) automatically load these object files in the proper order.

NAME

lex – generate programs for simple lexical tasks

SYNOPSIS

lex [**–rctvn**] [file] ...

DESCRIPTION

The *lex* command generates programs to be used in simple lexical analysis of text.

The input *files* (standard input default) contain strings and expressions to be searched for, and C text to be executed when strings are found.

A file **lex.yy.c** is generated which, when loaded with the library, copies the input to the output except when a string specified in the file is found; then the corresponding program text is executed. The actual string matched is left in *yytext*, an external character array. Matching is done in order of the strings in the file. The strings may contain square brackets to indicate character classes, as in **[abx–z]** to indicate **a**, **b**, **x**, **y**, and **z**; and the operators *****, **+**, and **?** mean respectively any non-negative number of, any positive number of, and either zero or one occurrence of, the previous character or character class. The character **.** is the class of all ASCII characters except new-line. Parentheses for grouping and vertical bar for alternation are also supported. The notation *r{d,e}* in a rule indicates between *d* and *e* instances of regular expression *r*. It has higher precedence than /, but lower than *, ?, +, and concatenation. Thus **[a–zA–Z]+** matches a string of letters. The character ^ at the beginning of an expression permits a successful match only immediately after a new-line, and the character $ at the end of an expression requires a trailing new-line. The character / in an expression indicates trailing context; only the part of the expression up to the slash is returned in *yytext*, but the remainder of the expression must follow in the input stream. An operator character may be used as an ordinary symbol if it is within " symbols or preceded by \.

Three subroutines defined as macros are expected: **input()** to read a character; **unput(***c***)** to replace a character read; and **output(***c***)** to place an output character. They are defined in terms of the standard streams, but you can override them. The program generated is named **yylex()**, and the library contains a **main()** which calls it. The action REJECT on the right side of the rule causes this match to be rejected and the next suitable match executed; the function **yymore()** accumulates additional characters into the same *yytext*; and the function **yyless(***p***)** pushes back the portion of the string matched beginning at *p*, which should be between *yytext* and *yytext+yyleng*. The macros *input* and *output* use files **yyin** and **yyout** to read from and write to, defaulted to **stdin** and **stdout**, respectively.

Any line beginning with a blank is assumed to contain only C text and is copied; if it precedes %% it is copied into the external definition area of the **lex.yy.c** file. All rules should follow a %%, as in YACC. Lines preceding %% which begin with a non-blank character define the string on the left to be the remainder of the line; it can be called out later by surrounding it with {}. Note that curly brackets do not imply parentheses; only string substitution is done.

EXAMPLE

```
D         [0–9]
%%
if        printf("IF statement\n");
[a–z]+    printf("tag, value %s\n",yytext);
0{D}+     printf("octal number %s\n",yytext);
{D}+      printf("decimal number %s\n",yytext);
"++"      printf("unary op\n");
"+"       printf("binary op\n");
"/*"        skipcommnts();
%%
 skipcommnts()
 {
          for (;;)
          {
                    while (input() != '*')
                              ;
                    if (input() != '/') {
                              unput(yytext[yyleng-1]);
                    } else
                              return;
          }
 }
```

The external names generated by *lex* all begin with the prefix yy or YY.

The flags must appear before any files. The flag **–r** indicates RATFOR actions, **–c** indicates C actions and is the default, **–t** causes the **lex.yy.c** program to be written instead to standard output, **–v** provides a one-line summary of statistics, **–n** will not print out the **–v** summary. Multiple files are treated as a single file. If no files are specified, standard input is used.

Certain table sizes for the resulting finite state machine can be set in the definitions section:

%p *n*   number of positions is *n* (default 2500)

%n *n*	number of states is *n* (500)
%e *n*	number of parse tree nodes is *n* (1000)
%a *n*	number of transitions is *n* (2000)
%k *n*	number of packed character classes is *n* (1000)
%o *n*	size of output array is *n* (3000)

The use of one or more of the above automatically implies the −v option, unless the −n option is used.

SEE ALSO

yacc(1).
Programmer's Guide.

BUGS

The −r option is not yet fully operational.

NAME

line – read one line

SYNOPSIS

line

DESCRIPTION

line copies one line (up to a new-line) from the standard input and writes it on the standard output. It returns an exit code of 1 on EOF and always prints at least a new-line. It is often used within shell files to read from the user's terminal.

SEE ALSO

sh(1).
read(2) in the *Programmer's Reference Manual*.

NAME

lint – a C program checker

SYNOPSIS

lint [option] ... file ...

DESCRIPTION

lint attempts to detect features of the C program files that are likely to be bugs, non-portable, or wasteful. It also checks type usage more strictly than the compilers. Among the things that are currently detected are unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions that return values in some places and not in others, functions called with varying numbers or types of arguments, and functions whose values are not used or whose values are used but none returned.

lint also attempts to diagnose printf/scanf statements for improper formatting.

The errors and warnings implied by the **−prototypes** option to *cc*(1) are in effect in *lint*. The prototype error and warning messages from *lint* contain less detail than *cc*(1) versions of these messages.

Arguments whose names end with **.c** are taken to be C source files. Arguments whose names end with **.ln** are taken to be the result of an earlier invocation of *lint* with either the **−c** or the **−o** option used. The **.ln** files are analogous to **.o** (object) files that are produced by the *cc*(1) command when given a **.c** file as input. Files with other suffixes are warned about and ignored.

Lint will take all the **.c**,**.ln**, and **llib-l***x***.ln** (specified by **−l***x*) files and process them in their command line order. By default, *lint* appends the standard C lint library (**llib-lc.ln**) to the end of the list of files. However, if the **−p** option is used, the portable C lint library (**llib-port.ln**) is appended instead. When the **−c** option is not used, the second pass of *lint* checks this list of files for mutual compatibility. When the **−c** option is used, the **.ln** and the **llib-l***x***.ln** files are ignored.

Any number of *lint* options may be used, in any order, intermixed with file-name arguments. The following options are used to suppress certain kinds of complaints:

−a Suppress complaints about assignments of long values to variables that are not long.

−b	Suppress complaints about **break** statements that cannot be reached. (Programs produced by *lex* or *yacc* will often result in many such complaints).
−h	Do not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.
−u	Suppress complaints about functions and external variables used and not defined, or defined and not used. (This option is suitable for running *lint* on a subset of files of a larger program).
−v	Suppress complaints about unused arguments in functions.
−x	Do not report variables referred to by external declarations but never used.

The following arguments alter *lint*'s behavior:

−l*x*	Include additional lint library **llib-l***x***.ln**. For example, you can include a lint version of the Math Library **llib-lm.ln** by inserting **−lm** on the command line. This argument does not suppress the default use of **llib-lc.ln**. These lint libraries must be in the assumed directory. This option can be used to reference local lint libraries and is useful in the development of multi-file projects.
−n	Do not check compatibility against either the standard or the portable lint library.
−p	Attempt to check portability to other dialects (IBM and GCOS) of C. Along with stricter checking, this option causes all non-external names to be truncated to eight characters and all external names to be truncated to six characters and one case.
−c	Cause *lint* to produce a **.ln** file for every **.c** file on the command line. These **.ln** files are the product of *lint*'s first pass only, and are not checked for inter-function compatibility.
−o lib	Cause *lint* to create a lint library with the name **llib-l***lib***.ln**. The **−c** option nullifies any use of the **−o** option. The lint library produced is the input that is given to *lint*'s second pass. The **−o** option simply causes this file to be saved in the named lint library. To produce a **llib-l***lib***.ln** without extraneous messages, use of the **−x** option is suggested. The **−v** option is useful if the source file(s) for the lint library are just external interfaces (for example, the way the file **llib-lc** is written). These option settings are also available through the use of ''lint comments'' (see below).

The **−D**, **−U**, and **−I** options of *cpp*(1) and the **−g** and **−O** options of *cc*(1) are also recognized as separate arguments. The **−g** and **−O** options are ignored, but, by recognizing these options, *lint*'s behavior is closer to that of the *cc*(1) command. Other options are warned about and ignored. The pre-processor symbol "lint" is defined to allow certain questionable code to be altered or removed for *lint*. Therefore, the symbol "lint" should be thought of as a reserved word for all code that is planned to be checked by *lint*.

Certain conventional comments in the C source will change the behavior of *lint*:

/*NOTREACHED*/
: at appropriate points stops comments about unreachable code. (This comment is typically placed just after calls to functions like *exit*(2)).

/*VARARGS*n**/
: suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0.

/*ARGSUSED*/
: turns on the **−v** option for the next function.

/*LINTLIBRARY*/
: at the beginning of a file shuts off complaints about unused functions and function arguments in this file. This is equivalent to using the **−v** and **−x** options.

Lint produces its first output on a per-source-file basis. Complaints regarding included files are collected and printed after all source files have been processed. Finally, if the **−c** option is not used, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source file name will be printed followed by a question mark.

The behavior of the **−c** and the **−o** options allows for incremental use of *lint* on a set of C source files. Generally, one invokes *lint* once for each source file with the **−c** option. Each of these invocations produces a **.ln** file which corresponds to the **.c** file, and prints all messages that are about just that source file. After all the source files have been separately run through *lint*, it is invoked once more (without the **−c** option), listing all the **.ln** files with the needed **−l***x* options. This will print all the inter-file inconsistencies. This scheme works well with *make*(1); it allows *make* to be used to *lint* only the source files that have been modified since the last time the set of source files were *lint*ed.

FILES

/usr/lib/lint	the directory where the lint libraries specified by the −l*x* option must exist
/usr/lib/lint/llib-lcurses.ln	declarations for C Library functions found in */usr/lib/libcurses.a* (binary format; source is in */usr/lib/lint/llib-lcurses.*
/usr/lib/lint/llib-lgl.ln	declarations for C Library functions found in */usr/lib/libgl.a* (binary format; source is in */usr/lib/lint/llib-lgl.*
/usr/lib/lint/lint[12]	first and second passes
/usr/lib/lint/llib-lc.ln	declarations for C Library functions (binary format; source is in */usr/lib/lint/llib-lc*)
/usr/lib/lint/llib-port.ln	declarations for portable functions (binary format; source is in */usr/lib/lint/llib-port*)
/usr/lib/lint/llib-lm.ln	declarations for Math Library functions (binary format; source is in */usr/lib/lint/llib-lm*)
*$TMPDIR/*lint**	temporaries (*/usr/tmp* used if the environment variable TMPDIR is not set)

SEE ALSO

cc(1), cpp(1), make(1).

BUGS

exit(2), *longjmp*(3C), and other functions that do not return are not understood; this causes various lies. *lint* does not understand the rules for arguments in functions with prototypes as well as it should.

NAME

List_tape – list the contents of a given backup tape

SYNOPSIS

List_tape [*–h hostname*]

DESCRIPTION

The *List_tape* command lists the contents of a given backup tape or any other "bru" format tape.

If a tape drive attached to a remote host is used, the name of the remote host needs to be specified with the **–h hostname** option on the command line. For remote listing to successfully work, the user should have a TCP/IP network connection to the remote host and also have "guest" login privileges on that host.

The List_tape command expects the tape to be in "bru" format.

SEE ALSO

Backup(1), Restore(1), bru(1).

NAME

loadmap – loads the colormap from a file

SYNOPSIS

loadmap file.map

DESCRIPTION

loadmap loads the current contents of the color map from a file. The color indices and color map entries must be in a file written by *savemap*(1G).

SEE ALSO

makemap(1g), savemap(1g)

NAME

listres - list resources in widgets

SYNOPSIS

listres [-option ...]

DESCRIPTION

The *listres* program generates a list of a widget's resource database. The class in which each resource is first defined, the instance and class name, and the type of each resource is listed.

OPTIONS

Listres accepts all of the standard toolkit command line options along with those listed below:

–known This option indicates that *listres* should print a two-column list of all of the widgets about which it has information. The name of the external widget class record variable is shown on the left and the superclass to subclass widget hierarchy is given on the right.

–nosuper
This option indicates that resources that are inherited from a superclass should not be listed. This is useful for determining which resources are new to a subclass.

–variable
This option indicates that widgets should be identified by the names of the class record variables rather than the class name given in the variable. This is useful for distinguishing subclasses that have the same class name as their superclasses.

–top *name*
This option specifies the name of the widget to be treated as the top of the hiearchy. Case is not significant, and the name may match either the class variable name or the class name. The default is ''core''.

–format *printf–string*
This option specifies the *printf*-style format string to be used to print out the name, instance, class, and type of each resource.

X DEFAULTS

To be written.

SEE ALSO

X(1), xrdb(1), appropriate widget documents

BUGS

On operating systems that do not support dynamic linking of run-time routines, this program must have all of its known widgets compiled in. The sources provide several tools for automating this process for various widget sets.

COPYRIGHT

AUTHOR

Jim Fulton, MIT X Consortium

NAME

login – sign on

SYNOPSIS

login [name [env-var ...]]

DESCRIPTION

The *login* command is used at the beginning of each terminal session and allows you to identify yourself to the system. It may be invoked as a command or by the system when a connection is first established. Also, it is invoked by the system when a previous user has terminated the initial shell by typing a *ctrl-d* to indicate an ''end-of-file.'' (See *How to Get Started* at the beginning of this volume for instructions on how to dial up initially.)

If *login* is invoked as a command it must replace the initial command interpreter. This is accomplished by typing:

```
exec login
```

from the initial shell.

Login asks for your user name (if not supplied as an argument), and, if appropriate, your password. Echoing is turned off (where possible) during the typing of your password, so it will not appear on the written record of the session.

Login reads **/etc/config/login.options** to determine site-specfic options. To change the options, the system administrator should create this file and insert the appropriate option keywords. The recognized options are:

passwdreq	# must have a password
maxtries=*number*	# max number of failed attempts allowed
disabletime=*seconds*	# amount of time to wait after maxtries
lastlog	# show last login info
syslog=all	# log successful and failed attempts to syslog
syslog=fail	# log failed attempts only

Keywords can be separated by whitespace or placed on separate lines. Keywords that take values must be one word with no whitespace between the keyword, equals sign and value. Anything after the # character is ignored, which is useful for disabling an option. Unrecognized words and blank lines are ignored, too.

In the default configuration, *login* will only prompt the user to enter a password if the specified user has a password in the system password file. If specified, the **passwdreq** option forces users who do not have passwords defined for them in the password file to choose a password before being allowed to login.

If the user name entered does not match any entry in the password file or the specified password is not the correct password for the user name, you will receive the message

Login incorrect

and a new login prompt will appear after a pause of several seconds. The **maxtries** option specifies the number of consecutive unsuccessful login attempts permitted before disabling the line. After **maxtries** attempts, further login attempts be disallowed for a period of time specified in seconds by the **disabletime** option.

Successful and unsuccessful login attempts can be logged to the *syslog*(3) facility if the **syslog** option is set. Log enties are written to the LOG_AUTH facility (see *syslog*(3) and *syslogd*(1M) for details).

The **lastlog** option enables *login* to inform the user of the last login attempt if the login is successful. It lists the most recent login date and the name of the terminal or remote host from which the previous login attempt occurred. All this login attempt information is recorded in the file **/usr/adm/lastlog/***username*. Individual users can inhibit the printing of these messages by creating a file called **.hushlogin** in their home directories.

At some installations, an option may be invoked that will require you to enter a second ''dialup'' password. This will occur only for dial-up connections, and will be prompted by the message ''dialup password:''. Both passwords are required for a successful login.

For remote logins over the network, *login* prints the contents of **/etc/issue** before prompting for a username or password. The file **/etc/nologin** disables remote logins if it exists; *login* prints the contents of this file before disconnecting the session.

The system may be configured to automate the login process after a system restart. When the file **/etc/autologin** exists and contains a valid user name, the system will log in as the specified user without prompting for a user name or password. The automatic login takes place only after a system restart; once the user logs out, the normal interactive login session will be used until the next restart. This is intended to be used at sites where the normal security mechanisms provided by *login* are not needed or desired.

If you do not complete the login successfully within a certain period of time (e.g., one minute), *login* silently disconnects the line.

On the graphics console, *login* is used only if the configuration variable *visuallogin* is turned off. See *pandora*(1) and *visuallogin*(4).

After a successful login, accounting files are updated, and if *sh*(1) is specified in the **/etc/passwd** file, the procedure **/etc/profile** is performed, the message-of-the-day, if any, is printed, the user-ID, the group-ID, the working directory, and the command interpreter (usually *sh*(1)) is initialized, and the file **.profile** in the working directory is executed, if it exists. These specifications are found in the **/etc/passwd** file entry for the user. The name of the command interpreter is – followed by the last component of the interpreter's path name (i.e., **–sh**). If this field in the password file is empty, then the default command interpreter, **/bin/sh** is used. If this field is "*", then the named directory becomes the root directory, the starting point for path searches for path names beginning with a **/**. At that point *login* is re-executed at the new level which must have its own root structure, including **/etc/login** and **/etc/passwd**.

If you are on logging in on the graphics console, then in addition to the above initialization, the graphics is initialized and the window system is started. If you do not wish to have the graphics started then you may type the string **NOGRAPHICS** after your login name.

The basic *environment* is initialized to:

> HOME=*your-login-directory*
> PATH=:/usr/sbin:/usr/bsd:/bin:/usr/bin:/usr/bin/X11
> SHELL=*last-field-of-passwd-entry*
> MAIL=/usr/mail/*your-login-name*
> LOGNAME=*your-login-name*
> USER=*your-login-name*
> TZ=*timezone-specification*
> TERM=*terminal-specification*

TZ and **TERM** are simply passed from *login's* environment to the new shell. The **PATH** environment variable is modified in the case of a superuser login to remove the current directory from the search path. The environment may be expanded or modified by supplying additional arguments to *login*, either at execution time or when *login* requests your login name. The arguments may take either the form *xxx* or *xxx=yyy*. Arguments without an equal sign are placed in the environment as

> L*n*=xxx

where *n* is a number starting at 0 and is incremented each time a new variable name is required. Variables containing an = are placed into the environment without modification. If they already appear in the environment, then they replace the older value. The variables **PATH, LOGNAME, IFS, MAIL, CDPATH, HOME,** and **SHELL** cannot be changed. This prevents people, logging into restricted shell environments, from spawning secondary shells which are not restricted. Up to 62 of these environment variables may be specified. Both *login* and *getty* understand simple single-character quoting conventions. Typing a backslash in front of a character quotes it and allows the inclusion of such things as spaces and tabs.

To enable dial-in line password protection, two files are required. The file **/etc/dialups** must contain of the name of any dialup ports (e.g., **/dev/ttyd2**) that require password protection. These are specified one per line. The second file, **/etc/d_passwd** consists of lines with the following format:

shell:*password*:

This file is scanned when the user logs in, and if the *shell* portion of any line matches the command interpreter that the user will get, the user is prompted for a password, which is encoded and compared to that specified in the *password* portion of the line.

NOTES

Autologin is controlled by the existence of the **/etc/autologin.on** file. The file is normally created at boot time to automate the login process and then removed by *login* to disable the autologin process for succeeding terminal sessions.

FILES

/etc/utmp	accounting
/etc/wtmp	accounting
/usr/mail/*username*	mailbox for user *username*
/etc/nologin	disable remote logins
/etc/issue	messages printed before login prompt
/etc/motd	message-of-the-day (successful logins only)
/etc/passwd	password file
/etc/cshrc	system-wide *csh*(1) commands
$HOME/.login	user's *csh*(1) commands for login shell
$HOME/.cshrc	user's *csh*(1) commands for every shell
/etc/profile	system profile (for *sh*(1))
$HOME/.profile	user's login profile (for *sh*(1))
/etc/autologin	autologin user identity
/etc/autologin.on	enable autologin
/etc/config/login.options	site-specific options
$HOME/.hushlogin	makes last login quieter
/usr/adm/lastlog/*username*	time of last login by *username*

SEE ALSO

csh(1), mail(1), newgrp(1), pandora(1), rlogin(1), sh(1), telnet(1), su(1M) in the *IRIX User's Reference Manual.*
passwd(4), profile(4), cshrc(4) visuallogin(4), environ(5) in the *IRIX Programmer's Reference Manual.*

DIAGNOSTICS

Login incorrect if the user name or the password cannot be matched.

No shell or *no directory* if the information in the password file entry that specifies the home directory or login shell for the selected user is not valid. Ask the system administrator to check the password file entry.

No utmp entry. You must exec "login" from the lowest level "sh" if you attempted to execute *login* as a command without using the shell's *exec* internal command or from other than the initial shell.

NAME

logname – get login name

SYNOPSIS

logname

DESCRIPTION

logname prints the name of the user who invoked the command. It gets this name via *cuserid*(3S).

SEE ALSO

su(1), login(1).
cuserid(3S), environ(5) in the *Programmer's Reference Manual*.

DIAGNOSTICS

logname exits with a value of 1 if *cuserid* returns NULL, otherwise it exits with a 0.

NAME

lorder – find ordering relation for an object library

SYNOPSIS

lorder file ...

DESCRIPTION

The input is one or more object or library archive *files* [see *ar*(1)]. The standard output is a list of pairs of object file or archive member names, meaning that the first file of the pair refers to external identifiers defined in the second. The output may be processed by *tsort*(1) to find an ordering of a library suitable for one-pass access by *ld*(1). Note that the link editor *ld*(1) is capable of multiple passes over an archive in the portable archive format [see *ar*(4)] and does not require that *lorder*(1) be used when building an archive. The usage of the *lorder*(1) command may, however, allow for a slightly more efficient access of the archive during the link edit process.

The following example builds a new library from existing **.o** files.

```
ar –cr library `lorder *.o | tsort`
```

FILES

TMPDIR/*symref	temporary files
TMPDIR/*symdef	temporary files

TMPDIR is usually /usr/tmp but can be redefined by setting the environment variable **TMPDIR** [see *tempnam*() in *tmpnam*(3S)].

SEE ALSO

ar(1), ld(1), tsort(1), ar(4).

CAVEAT

lorder will accept as input any object or archive file, regardless of its suffix, provided there is more than one input file. If there is but a single input file, its suffix must be **.o**.

NAME

lp, cancel – send/cancel requests to an LP line printer

SYNOPSIS

lp [**–c**] [**–d**dest] [**–m**] [**–n**number] [**–o**option] [**–s**] [**–t**title] [**–w**] files
cancel [ids] [printers]

DESCRIPTION

lp arranges for the named files and associated information (collectively called a *request*) to be printed by a line printer. If no file names are mentioned, the standard input is assumed. The file name – stands for the standard input and may be supplied on the command line in conjunction with named *files*. The order in which *files* appear is the same order in which they will be printed.

lp associates a unique *id* with each request and prints it on the standard output. This *id* can be used later to cancel (see *cancel*) or find the status (see *lpstat*(1)) of the request.

The following options to *lp* may appear in any order and may be intermixed with file names:

–c Make copies of the *files* to be printed immediately when *lp* is invoked. Normally, *files* will not be copied, but will be linked whenever possible. If the **–c** option is not given, then the user should be careful not to remove any of the *files* before the request has been printed in its entirety. It should also be noted that in the absence of the **–c** option, any changes made to the named *files* after the request is made but before it is printed will be reflected in the printed output.

–d*dest* Choose *dest* as the printer or class of printers that is to do the printing. If *dest* is a printer, then the request will be printed only on that specific printer. If *dest* is a class of printers, then the request will be printed on the first available printer that is a member of the class. Under certain conditions (printer unavailability, file space limitation, etc.), requests for specific destinations may not be accepted (see *accept*(1M) and *lpstat*(1)). By default, *dest* is taken from the environment variable **LPDEST** (if it is set). Otherwise, a default destination (if one exists) for the computer system is used. Destination names vary between systems (see *lpstat*(1)).

–m Send mail (see *mail*(1)) after the files have been printed. By default, no mail is sent upon normal completion of the print request.

−n*number*	Print *number* copies (default of 1) of the output.
−o*option*	Specify printer-dependent or class-dependent *options*. Several such *options* may be collected by specifying the **−o** keyletter more than once. For more information about what is valid for *options*, see **Models** in *lpadmin*(1M).
−s	Suppress messages from *lp*(1) such as "request id is ...".
−t*title*	Print *title* on the banner page of the output.
−w	Write a message on the user's terminal after the *files* have been printed. If the user is not logged in, then mail will be sent instead.

Cancel cancels line printer requests that were made by the *lp*(1) command. The command line arguments may be either request *ids* (as returned by *lp*(1)) or *printer* names (for a complete list, use *lpstat*(1)). Specifying a request *id* cancels the associated request even if it is currently printing. Specifying a *printer* cancels the request which is currently printing on that printer. In either case, the cancellation of a request that is currently printing frees the printer to print its next available request.

FILES

/usr/spool/lp/*

SEE ALSO

enable(1), lpstat(1), mail(1).
accept(1M), lpadmin(1M), lpsched(1M) in the *System Administrator's Reference Manual.*

NAME

lpq – spool queue examination program

SYNOPSIS

lpq [-l] [–Pprinter] [job # ...] [user ...]

DESCRIPTION

lpq examines the spooling area used by *lpd*(1M) for printing files on the line printer, and reports the status of the specified jobs or all jobs associated with a user. *lpq* invoked without any arguments reports on any jobs currently in the queue. A –P flag may be used to specify a particular printer, otherwise the default line printer is used (or the value of the PRINTER variable in the environment). All other arguments supplied are interpreted as user names or job numbers to filter out only those jobs of interest.

For each job submitted (i.e. invocation of *lpr*(1)) *lpq* reports the user's name, current rank in the queue, the names of files comprising the job, the job identifier (a number which may be supplied to *lprm*(1) for removing a specific job), and the total size in bytes. The –l option causes information about each of the files comprising the job to be printed. Normally, only as much information as will fit on one line is displayed. Job ordering is dependent on the algorithm used to scan the spooling directory and is supposed to be FIFO (First in First Out). File names comprising a job may be unavailable (when *lpr*(1) is used as a sink in a pipeline) in which case the file is indicated as ''(standard input)".

If *lpq* warns that there is no daemon present (i.e. due to some malfunction), the *lpc*(1) command can be used to restart the printer daemon.

BUGS

Due to the dynamic nature of the information in the spooling directory lpq may report unreliably. Output formatting is sensitive to the line length of the terminal; this can results in widely spaced columns.

DIAGNOSTICS

Unable to open various files. The lock file being malformed. Garbage files when there is no daemon active, but files in the spooling directory.

FILES

/etc/termcap	for manipulating the screen for repeated display
/etc/printcap	to determine printer characteristics
/usr/spool/*	the spooling directory, as determined from printcap
/usr/spool/*/cf*	control files specifying jobs
/usr/spool/*/lock	the lock file to obtain the currently active job

SEE ALSO

lpr(1), lprm(1), lpc(1M), lpd(1M)

NAME

lpr – off line print

SYNOPSIS

lpr [**–P***printer*] [**–#***num*] [**–C** *class*] [**–J** *job*] [**–T** *title*] [**–i** [*numcols*]] [**–1234** *font*] [**–w***num*] [**–pltndgvcfrmhs**] [name ...]

DESCRIPTION

Lpr uses a spooling daemon to print the named files when facilities become available. If no names appear, the standard input is assumed. The **–P** option may be used to force output to a specific printer. Normally, the default printer is used (site dependent), or the value of the environment variable PRINTER is used.

The following single letter options are used to notify the line printer spooler that the files are not standard text files. The spooling daemon will use the appropriate filters to print the data accordingly.

–p Use *pr*(1) to format the files (equivalent to *print*).

–l Use a filter which allows control characters to be printed and suppresses page breaks.

–t The files are assumed to contain data from *troff*(1) (cat phototypesetter commands).

–n The files are assumed to contain data from *ditroff* (device independent troff).

–d The files are assumed to contain data from *tex*(l) (DVI format from Stanford).

–g The files are assumed to contain standard plot data as produced by the *plot*(3X) routines (see also *plot*(1G) for the filters used by the printer spooler).

–v The files are assumed to contain a raster image for devices like the Benson Varian.

–c The files are assumed to contain data produced by *cifplot*(l).

–f Use a filter which interprets the first character of each line as a standard FORTRAN carriage control character.

The remaining single letter options have the following meaning.

–r Remove the file upon completion of spooling or upon completion of printing (with the –s option).

–m Send mail upon completion.

−h Suppress the printing of the burst page.

−s Use symbolic links. Usually files are copied to the spool directory.

The −C option takes the following argument as a job classification for use on the burst page. For example,

```
lpr −C EECS foo.c
```

causes the system name (the name returned by *hostname*(1)) to be replaced on the burst page by EECS, and the file foo.c to be printed.

The −J option takes the following argument as the job name to print on the burst page. Normally, the first file's name is used.

The −T option uses the next argument as the title used by *pr*(1) instead of the file name.

To get multiple copies of output, use the *−#num* option, where *num* is the number of copies desired of each file named. For example,

```
lpr −#3 foo.c bar.c more.c
```

would result in 3 copies of the file foo.c, followed by 3 copies of the file bar.c, etc. On the other hand,

```
cat foo.c bar.c more.c | lpr −#3
```

will give three copies of the concatenation of the files.

The −i option causes the output to be indented. If the next argument is numeric, it is used as the number of blanks to be printed before each line; otherwise, 8 characters are printed.

The −w option takes the immediately following number to be the page width for *pr*.

The −s option will use *symlink*(2) to link data files rather than trying to copy them so large files can be printed. This means the files should not be modified or removed until they have been printed.

The option −1234 Specifies a font to be mounted on font position *i*. The daemon will construct a *.railmag* file referencing */usr/lib/vfont/name.size.*

DIAGNOSTICS

If you try to spool too large a file, it will be truncated. *Lpr* will object to printing binary files. If a user other than root prints a file and spooling is disabled, *lpr* will print a message saying so and will not put jobs in the queue. If a connection to *lpd* on the local machine cannot be made, *lpr* will say that the daemon cannot be started. Diagnostics may be printed in the daemon's log file regarding missing spool files by *lpd*.

BUGS

Fonts for *troff* and *tex* reside on the host with the printer. It is currently not possible to use local font libraries.

FILES

/etc/passwd	personal identification
/etc/printcap	printer capabilities data base
/usr/lib/lpd*	line printer daemons
/usr/spool/*	directories used for spooling
/usr/spool/*/cf*	daemon control files
/usr/spool/*/df*	data files specified in "cf" files
/usr/spool/*/tf*	temporary copies of "cf" files

SEE ALSO

lpq(1), lprm(1), pr(1), symlink(2), printcap(4), lpc(1M), lpd(1M)

NAME

lprm – remove jobs from the line printer spooling queue

SYNOPSIS

lprm [–**P***printer*] [–] [job # ...] [user ...]

DESCRIPTION

Lprm will remove a job, or jobs, from a printer's spool queue. Since the spooling directory is protected from users, using *lprm* is normally the only method by which a user may remove a job.

Lprm without any arguments will delete the currently active job if it is owned by the user who invoked *lprm*.

If the – flag is specified, *lprm* will remove all jobs which a user owns. If the super-user employs this flag, the spool queue will be emptied entirely. The owner is determined by the user's login name and host name on the machine where the *lpr* command was invoked.

Specifying a user's name, or list of user names, will cause *lprm* to attempt to remove any jobs queued belonging to that user (or users). This form of invoking *lprm* is useful only to the super-user.

A user may dequeue an individual job by specifying its job number. This number may be obtained from the *lpq*(1) program, e.g.

```
% lpq –l

1st: ken                        [job #013ucbarpa]
     (standard input)           100 bytes
% lprm 13
```

Lprm will announce the names of any files it removes and is silent if there are no jobs in the queue which match the request list.

Lprm will kill off an active daemon, if necessary, before removing any spooling files. If a daemon is killed, a new one is automatically restarted upon completion of file removals.

The –P option may be usd to specify the queue associated with a specific printer (otherwise the default printer, or the value of the PRINTER variable in the environment is used).

DIAGNOSTICS

''Permission denied" if the user tries to remove files other than his own.

BUGS

Since there are race conditions possible in the update of the lock file, the currently active job may be incorrectly identified.

FILES

/etc/printcap	printer characteristics file
/usr/spool/*	spooling directories
/usr/spool/*/lock	lock file used to obtain the pid of the current daemon and the job number of the currently active job

SEE ALSO

lpr(1), lpq(1), lpd(1M)

NAME

lpstat – print LP status information

SYNOPSIS

lpstat [options]

DESCRIPTION

lpstat prints information about the current status of the LP spooling system.

If no *options* are given, then *lpstat* prints the status of all requests made to *lp*(1) by the user. Any arguments that are not *options* are assumed to be request *ids* (as returned by *lp*). *lpstat* prints the status of such requests. *Options* may appear in any order and may be repeated and intermixed with other arguments. Some of the keyletters below may be followed by an optional *list* that can be in one of two forms: a list of items separated from one another by a comma, or a list of items enclosed in double quotes and separated from one another by a comma and/or one or more spaces. For example:

```
-u"user1, user2, user3"
```

The omission of a *list* following such keyletters causes all information relevant to the keyletter to be printed, for example:

```
lpstat -o
```

prints the status of all output requests.

–a[*list*]	Print acceptance status (with respect to *lp*) of destinations for requests. *List* is a list of intermixed printer names and class names.
–c[*list*]	Print class names and their members. *List* is a list of class names.
–d	Print the system default destination for *lp*.
–o[*list*]	Print the status of output requests. *List* is a list of intermixed printer names, class names, and request *ids*.
–p[*list*]	Print the status of printers. *List* is a list of printer names.
–r	Print the status of the LP request scheduler
–s	Print a status summary, including the system default destination, a list of class names and their members, and a list of printers and their associated devices.
–t	Print all status information.
–u[*list*]	Print status of output requests for users. *List* is a list of login names.

−v[*list*] Print the names of printers and the path names of the devices associated with them. *List* is a list of printer names.

FILES

/usr/spool/lp/*

SEE ALSO

enable(1), lp(1).

NAME

lptest – generate lineprinter ripple pattern

SYNOPSIS

lptest [length [count]]

DESCRIPTION

Lptest writes the traditional "ripple test" pattern on standard output. In 96 lines, this pattern will print all 96 printable ASCII characters in each position. While originally created to test printers, it is quite useful for testing terminals, driving terminal ports for debugging purposes, or any other task where a quick supply of random data is needed.

The *length* argument specifies the output line length if the the default length of 79 is inappropriate.

The *count* argument specifies the number of output lines to be generated if the default count of 200 is inappropriate. Note that if *count* is to be specified, *length* must be also be specified.

NAME

ls – list contents of directory

SYNOPSIS

ls [**–RaAdCLHxmlnogrtucpFbqisf**] [names]

DESCRIPTION

For each directory argument, *ls* lists the contents of the directory; for each file argument, *ls* repeats its name and any other information requested. The output is sorted alphabetically by default. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments appear before directories and their contents.

There are three major listing formats. The default format is to list one entry per line, the **–C** and **–x** options enable multi-column formats, and the **–m** option enables stream output format. In order to determine output formats for the **–C**, **–x**, and **–m** options, *ls* uses an environment variable, **COLUMNS**, to determine the number of character positions available on one output line. If this variable is not set, the *terminfo*(4) database is used to determine the number of columns, based on the environment variable **TERM**. If this information cannot be obtained, 80 columns are assumed.

The *ls* command has the following options:

–R	Recursively list subdirectories encountered.
–a	List all entries, including those that begin with a dot (**.**), which are normally not listed.
–A	Like **–a** except it does not list the **.** and **..** directories.
–d	If an argument is a directory, list only its name (not its contents); often used with **–l** to get the status of a directory.
–C	Multi-column output with entries sorted down the columns.
–L	If the file is a symbolic link, list the file that the link references.
–H	If the file is a symbolic link, list the file itself. This is the default behavior.
–x	Multi-column output with entries sorted across rather than down the page.
–m	Stream output format; files are listed across the page, separated by commas.
–l	List in long format, giving mode, number of links, owner, group, size in bytes, and time of last modification for each file (see below). If the file is a special file, the size field will instead contain the major and minor device numbers rather than a size.

−n The same as **−l**, except that the owner's UID and group's GID numbers are printed, rather than the associated character strings.

−o The same as **−l**, except that the group is not printed.

−g The same as **−l**, except that the owner is not printed.

−r Reverse the order of sort to get reverse alphabetic or oldest first as appropriate.

−t Sort by time stamp (latest first) instead of by name. The default is the last modification time. (See **−u** and **−c**.)

−u Use time of last access instead of last modification for sorting (with the **−t** option) or printing (with the **−l** option).

−c Use time of last modification of the i-node (file created, mode changed, etc.) for sorting (**−t**) or printing (**−l**).

−p Put a slash (/) after the name of each file that is a directory.

−F Put a slash (/) after the name of each file that is a directory, put an asterisk (*) after the name of each file that is executable, put a commercial at sign (@) after the name of each file that is a symbolic link and put an equals sign (=) after the name of each file that is a AF_UNIX address family socket.

−b Force printing of non-printable characters to be in the octal **\ddd** notation.

−q Force printing of non-printable characters in file names as the character question mark (?).

−i For each file, print the i-number in the first column of the report.

−s Give size in blocks, including indirect blocks, for each entry.

−f Force each argument to be interpreted as a directory and list the name found in each slot. This option turns off **−l**, **−t**, **−s**, and **−r**, and turns on **−a**; the order is the order in which entries appear in the directory.

The mode printed under the **−l** option consists of ten characters. The first character may be one of the following:

d	the entry is a directory;
b	the entry is a block special file;
c	the entry is a character special file;
l	the entry is a symbolic link;
p	the entry is a fifo (a.k.a. "named pipe") special file;

s	the entry is a AF_UNIX address family socket;
–	the entry is an ordinary file.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to the owner's permissions; the next to permissions of others in the user-group of the file; and the last to all others. Within each set, the three characters indicate permission to read, to write, and to execute the file as a program, respectively. For a directory, "execute" permission is interpreted to mean permission to search the directory for a specified file.

ls –l (the long list) prints its output as follows:

```
–rwxrwxrwx   1 smith   dev      10876   May  16 9:42 part2
```

This horizontal configuration provides a good deal of information. Reading from right to left, you see that the current directory holds one file, named "part2." Next, the last time that file's contents were modified was 9:42 A.M. on May 16. The file is moderately sized, containing 10,876 characters, or bytes. The owner of the file, or the user, belongs to the group "dev" (perhaps indicating "development"), and his or her login name is "smith." The number, in this case "1," indicates the number of links to file "part2." Finally, the row of dash and letters tell you that user, group, and others have permissions to read, write, execute "part2."

The execute (x) symbol here occupies the third position of the three-character sequence. A – in the third position would have indicated a denial of execution permissions.

The permissions are indicated as follows:

r	the file is readable
w	the file is writable
x	the file is executable
–	the indicated permission is *not* granted
l	mandatory locking will occur during access (the set-group-ID bit is on and the group execution bit is off)
s	the set-user-ID or set-group-ID bit is on, and the corresponding user or group execution bit is also on
S	undefined bit-state (the set-user-ID bit is on and the user execution bit is off)
t	the 1000 (octal) bit, or sticky bit, is on (see *chmod*(1)), and execution is on
T	the 1000 bit is turned on, and execution is off (undefined bit-state)

For user and group permissions, the third position is sometimes occupied by a character other than **x** or **–**. **s** also may occupy this position, referring to the state of the set-ID bit, whether it be the user's or the group's. The ability to assume the same ID as the user during execution is, for example, used during login when you begin as root but need to assume the identity of the user stated at "login."

In the case of the sequence of group permissions, **l** may occupy the third position. **l** refers to mandatory file and record locking. This permission describes a file's ability to allow other files to lock its reading or writing permissions during access.

For others permissions, the third position may be occupied by **t** or **T**. These refer to the state of the sticky bit and execution permissions.

EXAMPLES

An example of a file's permissions is:

```
–rwxr—r—
```

This describes a file that is readable, writable, and executable by the user and readable by the group and others.

Another example of a file's permissions is:

```
–rwsr–xr–x
```

This describes a file that is readable, writable, and executable by the user, readable and executable by the group and others, and allows its user-ID to be assumed, during execution, by the user presently executing it.

Another example of a file's permissions is:

```
–rw–rwl——
```

This describes a file that is readable and writable only by the user and the group and can be locked during access.

An example of a command line:

```
ls –a
```

This command will print the names of all files in the current directory, including those that begin with a dot (.), which normally do not print.

Another example of a command line:

```
ls –aisn
```

This command will provide you with quite a bit of information including all files, including non-printing ones (**a**), the **i**-number—the memory address of the i-node associated with the file—printed in the left-hand column (**i**); the size (in blocks) of the files, printed in the column to the right of the i-numbers (**s**); finally, the report is displayed in the numeric version of the long list, printing the UID (instead of user name) and GID (instead of group name) numbers associated with the files.

When the sizes of the files in a directory are listed, a total count of blocks, including indirect blocks, is printed.

FILES

/etc/passwd	user IDs for **ls −l** and **ls −o**
/etc/group	group IDs for **ls −l** and **ls −g**
/usr/lib/terminfo/?/*	terminal information database

SEE ALSO

chmod(1), find(1).

BUGS

Unprintable characters in file names may confuse the columnar output options.